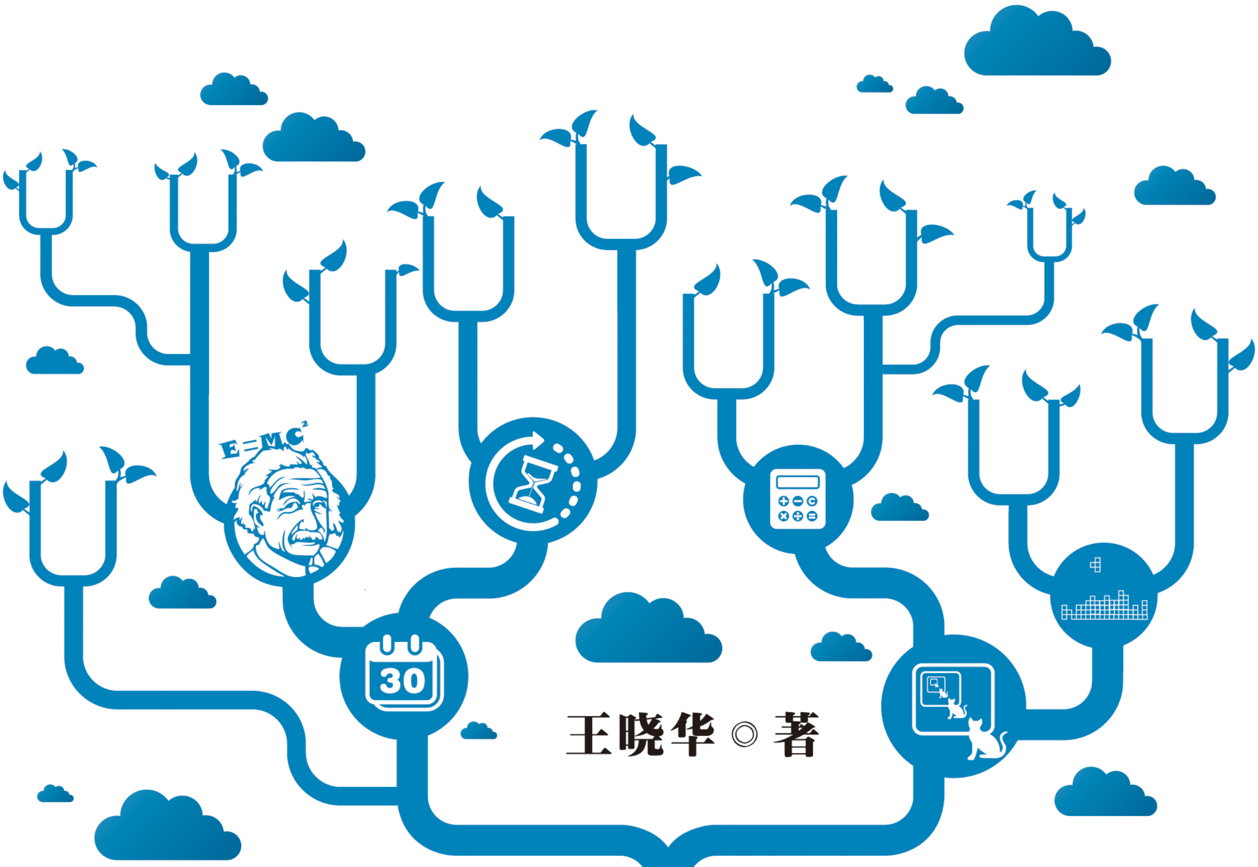
图灵原创



王晓华◎著

淋漓尽致展现算法本质
广泛涵盖常用算法结构及应用

算法领域超人气著作
一本书玩转算法，尽享算法乐趣

算法的乐趣

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

王晓华

2005年毕业于华中科技大学，目前在中兴通讯上海研发中心从事光纤接入网通讯设备开发，担任EPON（以太网无源光网络）业务软件开发经理，参与开发的PON设备在全球部署过亿线，为数亿家庭提供宽带接入服务。

业余时间喜欢研究算法和写作博客，最大的乐趣就是用程序解决生活中的问题：

★为了方便使用Visual Studio 6.0开发软件，曾特意编写并开源了一个tabbar插件；

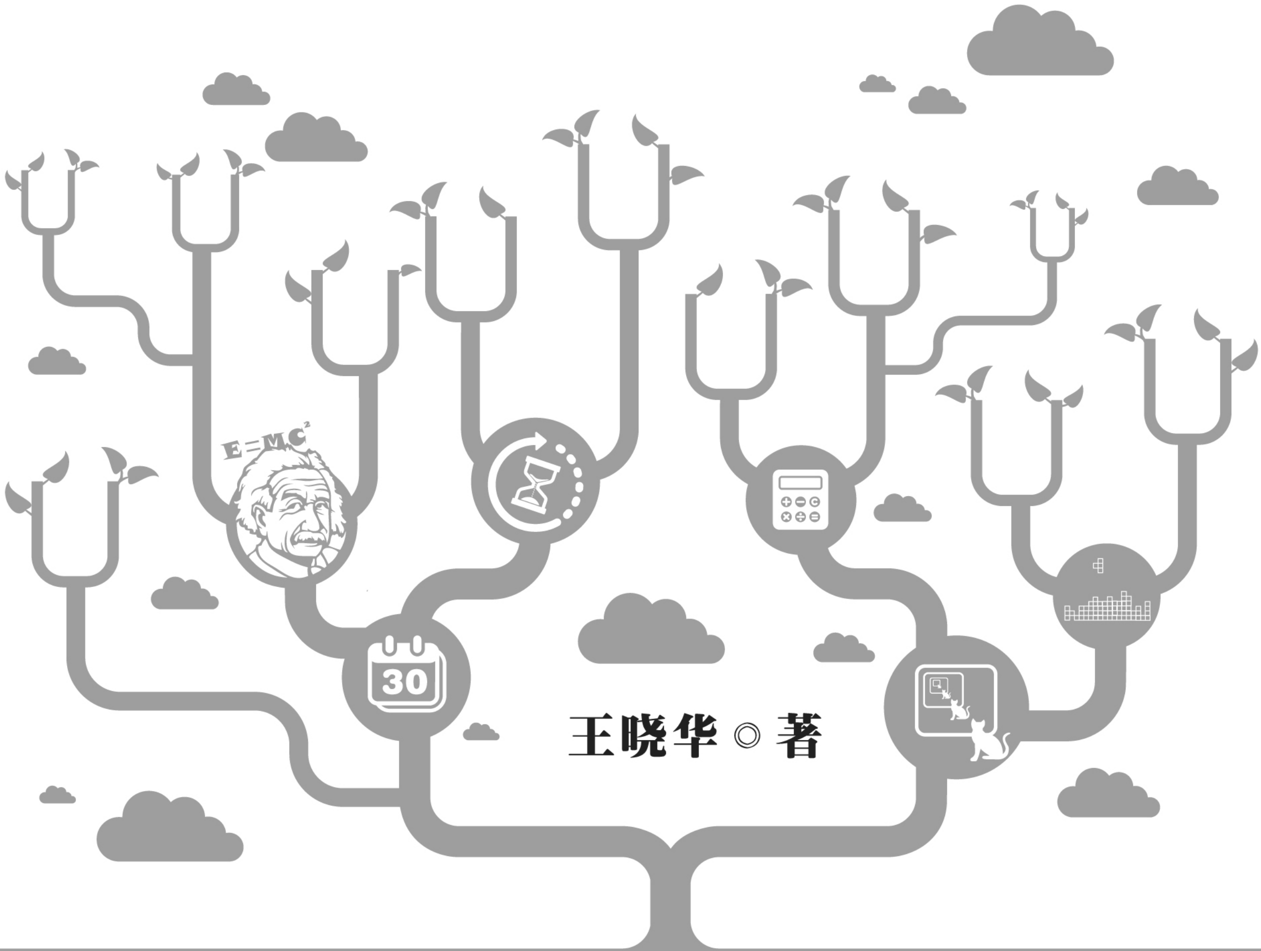
★为了文档安全，开发了一个基于layerFSD技术的透明文件加密系统；

★使用Source Insight软件觉得不习惯，于是以外挂的形式开发了TabSiPlus插件……

算法可以做的事情还有很多，期待我们会有更多发现！

作者博客

图灵原创



王晓华 ◎ 著

算法的乐趣

人民邮电出版社
北京

图书在版编目（C I P）数据

算法的乐趣 / 王晓华著. -- 北京 : 人民邮电出版
社, 2015.4（2017.1重印）
（图灵原创）
ISBN 978-7-115-38537-6

Ⅰ. ①算… Ⅱ. ①王… Ⅲ. ①算法语言－程序设计
Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第033951号

内 容 提 要

本书从一系列有趣的生活实例出发，全面介绍了构造算法的基础方法及其广泛应用，生动地展现了算法的趣味性和实用性。全书分为两个部分，第一部分介绍了算法的概念、常用的算法结构以及实现方法，第二部分介绍了算法在各个领域的应用，如物理实验、计算机图形学、数字音频处理等。其中，既有各种大名鼎鼎的算法，如神经网络、遗传算法、离散傅里叶变换算法及各种插值算法，也有不起眼的排序和概率计算算法。讲解浅显易懂而不失深度和严谨，对程序员有很大的启发意义。书中所有的示例都与生活息息相关，淋漓尽致地展现了算法解决问题的本质，让你爱上算法，乐在其中。

本书适合软件开发人员、编程和算法爱好者以及计算机专业的学生阅读。

◆ 著　　　　王晓华
责任编辑　王军花
执行编辑　张　霞
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：26.25
字数：614千字　　　　2015年4月第1版
印数：13 801－14 800册　　2017年1月北京第7次印刷

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

序　一

读《算法的乐趣》的乐趣超出了我的预料。

说到算法，大部分计算机专业的同学的第一反应估计是MIT出版社的经典教材《算法导论》（*Introduction to Algorithms*）。这是一本由浅入深的好书，堪称“神书”——别看书挺厚，但是对初学者来说很难弄懂的问题也娓娓道来，让人看一遍就明白；而且作者用最简单的英语词汇和句法写书，以至于世界各地的学生们，不需要英语很好，即可读懂原版。只是看完这本大部头之后，总有一些意犹未尽的感觉——对我们日常生活中常见的比如音乐播放器里以及电子游戏里的算法并没有太多介绍。而这些正是《算法的乐趣》中主要的部分。

在Amazon上，另外两本排名靠前的经典算法教材是Jon Kleinberg的《算法设计》（*Algorithm Design*）和Steven S. Skiena的《算法设计手册》（*The Algorithm Design Manual*）。这两本出自名家之手的教材和很多教材一样，按照算法的类型或者背后的设计思路来组织内容。这是教材应该做的，“授人以鱼不如授人以渔”，传授思路而不是算法本身是教材的写作目的。可是算法最有意思的地方首先在于算法本身，因为算法是为了解决实际问题而设计的，所以让大家认识到算法奥妙的自然顺序应该是先展示有趣的问题，再展示优雅的算法，最后归纳设计思路。而这正是《算法的乐趣》吸引人的地方。

说到乐趣，总让我想起我学习和使用数学知识的经历。虽然我的学位是关于统计机器学习的，而且毕业后一直从事相关工作，但是我从小学一年级到博士第三年都对数学毫无兴趣，因为学校的老师和数学成绩好的同学都说不明白数学的用处，以至于我一直以为数学的作用只是锻炼和展示自己的聪明，博得老师的表扬，成为陈景润那样为国争光的英雄。而这些对我实在没有吸引力，而且我认为恐怕对绝大部分学生都没什么吸引力。

我认识到数学的价值，是因为在博士第三年把研究方向换到了统计机器学习。在读教材的时候，我曾想验证“数学无用”，所以费尽心力地试图写一个程序来判断一个64×64像素的图片里到底是数字“1”还是数字“9”，却发现无论如何也很难写一个有效的程序；可是利用教材里的数学知识却能设计和“训练”一个数学模型，准确地识别任意字符。因为体会到了数学的用处，我兴奋地用了一年的时间复习大学本科的数学课程，然后才读懂了人工智能的专业教材和论文。此后才有所创新，发表论文，到博士毕业。这整个过程用了三年，而效果超过了之前19年数学教育的效果。

在这个过程中，我自然而然地开始注意数学知识的前因（比如为什么人们会关注长度、面积，怎么会有人考虑勾股定理这样的规律）以及后果（今天的数学知识能给物理学和机器智能带来什么样的帮助），也开始归纳和了解各种数学系统背后的规律，能体会哥德尔定理阐述的意思。当然，也破除了“数学是各种科学之母”之类的迷信，数学当然不是“科学之母”，而是“科学之子”，是先有物理学、力学和天文学，才有的数学；先有应用场景后有工具，先有探索后有归纳。

算法也是如此。先有工程问题需要解决，算法是解法，设计算法是寻求解法。虽然算法作为一门科学是归纳寻求解法的思路，但学习这种归纳法的前提是能体会各种具体算法的用处和效果。意识到这一点，自然也就破除了诸如“学好数学才能学好算法”之类的迷信。而把算法解决的各种有趣问题罗列出来，把算法的可爱之处展示给愿意发现和体会生活中点滴乐趣的读者们，正是《算法的乐趣》在技术价值之外的一层社会价值。

十年前，当我们坐在课堂里学习算法的时候，我们学到的是如何用人脑寻求解法，然后把解法写成程序，让计算机照着执行去解决问题。这是“经典算法”。最近十几年，随着Internet产业的兴起，Internet服务在不断取代原来由人提供的服务，这就要求机器拥有一定程度上能取代人的“智能”。在搜索引擎、推荐系统和广告系统等各个领域里，类似上述“识别数字”的问题越来越多，而人工智能和机器学习的应用也越来越深入我们的生活。机器学习算法的设计目标和“经典算法”不同——不是让人来想解法，而是让计算机从数据归纳知识——有了这些知识，计算机就能自己寻求解法。

虽然经典算法和机器学习算法之间的差别大得如同一场革命，但是由经典而入机器学习的过程却是自然而然的。比如《算法的乐趣》中介绍的曲线拟合问题，就是supervised learning（有监督学习），而音乐播放器里常用的傅里叶变换和其他时域频域变换则是unsupervised learning（无监督学习）的技术基础，棋类游戏算法是博弈论和reinforcement learning（强化学习）的经典例子。我常见有朋友从读数学教材开始探索机器学习和人工智能算法，也常看到有人不堪忍受长时间缺乏乐趣的探索以至于半途而废。如果是这样，也许不如从《算法的乐趣》开始这个探索过程。

我曾经以为从乐趣出发阐述算法的书会从西方发芽，没想到先看到了一本中文书。这真超出了我的预料。

王益
LinkedIn高级主任分析师

序　　二

当图灵出版社的编辑找到我希望我为这本书写个序的时候，我和旁边的同事调侃了一句：又一本简版的《算法导论》要诞生了。但是我还是下载了附件阅读了这本书，当翻到目录的时候，我的兴趣就被燃起来了，转头和同事说，也许这是一本不错的书。

程序员到底需不需要学习算法？这个问题被争论的次数绝对不亚于"Java是不是最好的语言""VIM和Emacs谁是最好的编辑器""程序员是不是需要学习数学"。为了避免陷入这样的争论里，我们先对"算法"一词做个转换定义，什么是算法？下面我举几个我亲身经历的例子。

有一次我们发布了一个APP，在注册时要求用户输入自己的真实姓名，但是粗心的工程师忘记了要求用户填写自己的性别，更可怜的是在欢迎页上面明晃晃地写着"欢迎XXX先生注册XX网"，可是应用已经发布到了App Store，到底怎么办？有一位工程师提出了一个办法，我们根据已有的用户姓名和性别作为训练集，来预测新用户到底是男还是女，为了让这个错误尽快得到修复，我们使用了最简单的朴素贝叶斯分类算法，最终测试集上的预测准确率达到93%，也就是说我们解决了93%用户的体验问题。我把这类算法称为专业类算法，也就是招聘网站上算法工程师要求的算法，例如图像处理工程师、数据挖掘工程师等。

有一次我们有一个相似性搜索的需求，数据量不大，只有几万条的数据记录，没有必要用ES这样的搜索引擎。例如输入"长沙市"， 也希望可以找到"我爱湖南长沙""沙市小吃"等，且不说这个需求是否合理，我们单纯来讨论这个问题的解决方案。工程师实现的第一版是将所有字的组合全部列举出来，然后在数据库里做like操作，性能无法接受。于是我们提出了另一种解决方案：将数据库中的每个词都拆成单字，做成集合，保存在缓存中。接下来只需要对集合做交集操作，以字为单位计算词与词之间的相似性，性能问题一下就解决了。这种解题思路在《编程珠玑》中屡见不鲜，这不足以称为具体的算法，几乎都是在梳理我们的逻辑，训练我们解决问题的能力，我把这类算法称为逻辑类算法，或者技巧类算法。

还有一次，我们有个需求是帮助用户做旅游的行程规划，其中的情况比较复杂，因为除了地理位置之外，还需要包含目的地的过往用户评价、所需耗时、不同城市的住宿花费等。但是如果我们仔细分析，可以基于产品设计去拆分问题。在在线部分，我们可以去使用基于路程的最短图路径算法，或者基于价格的贪心算法，也可以在综合排序处为用户选择使用了变形加权的最短图路径算法。在离线部分，由于图的节点和边都较少，可以使用穷举法来为用户找到几种不同类型的最优解。这些算法都是在《算法导论》和《数据结构》中有着详细讨论的算法，书中的每一个

算法和数据结构都是作者多年来抽象总结出的通用思路，我称之为通用类算法。

再说一个最基本的，我们做一个网站允许用户发布状态，在高峰期并发量太大，数据库不堪重负，所以我们需要将用户的插入记录先存入到消息队列中，保证用户的正常使用，然后再落地到MySQL数据库中。大家都会想到选择队列这样一种先入先出的数据结构，这也属于一种算法。

通过上面的几个例子，你会不会觉得你的身边处处都是算法？那么到底什么是算法？我们看看标准的定义：能够对一定规范的输入，在有限时间内获得所要求的输出的一系列指令都叫作算法。这个定义太抽象了，让我们简单来说，算法其实就是解决问题的思路和办法。那么从这一点来说，你还会说算法不重要么？

那么为什么还会有很多学生，甚至已经工作了很久的朋友还会说大学学的东西没有意义，算法没有用呢？归根结底是因为大家不知道为什么学，或者说缺乏算法的场景化。我在读大学的时候，经常做一些简单的网站，用到的技术无外乎是最简单的对数据库的增删改查，当时最大的感觉就是算法没有用。后来随着工作的深入，我开始逐步地意识到算法的重要性，逐渐地把算法捡了起来。

这本书给我最大的惊喜是没有像一般的算法书一样单纯地去讲算法和数据结构本身，那样无论语言多风趣，只要一谈到关键的问题也会马上变得无趣起来。作者在每一章都给出了一个实际的问题，然后尝试用算法去解决这一个问题，没有局限于通用类算法，而是同时涵盖逻辑类算法、通用类算法和专业类算法，真正是在训练读者解决问题的能力，而解决问题的能力，正是任何一家公司所需人才的最核心的技能。

另外，我已经在幻想作者在下一本书里可以把工作中的实际场景列举出来，更进一步地讲述工作中的算法，让每一个在校学生都可以意识到算法对于未来工作的重要性，也让每一位从业者拍案叫绝：“原来这个问题可以这样解！”让人人谈算法，人人写算法，引发软件行业的全民算法潮。

黄鑫（飞林沙）
极光推送首席科学家

序　三

如果说《啊哈！算法》是算法界的小白书，内容太少看得不过瘾，那么这本《算法的乐趣》或许可以带你一起牛逼一起飞。当我刚拿到书的目录的时候，我就很期待，因为终于有一本算法书可以系统地和大伙说一说这些我也很想与大伙说的伟大算法。

暴力盲目的搜索算法往往让计算机显得很笨甚至有点痴呆，如果你想设计一个“狡猾”的程序，那么本书中的搜索剪枝、A*寻径、博弈树以及遗传算法等将给你带来启发。快速傅里叶变换，这么霸气而又高大上的名字，其实在我们生活中的应用随处可见，家中的Wi-Fi、智能手机、电话、路由器等几乎所有内置计算机系统的东西都会以各种方式使用这个算法。RLE数据压缩算法，在文档、视频、音乐、数据存储、云计算、数据库等几乎所有应用中都有着广泛的运用。压缩算法令系统更有效，成本更低。再来说密码学算法中非常重要的RSA算法，如果没有这些算法，互联网就会变得不安全，电子交易就不会如此可信。

好玩的算法还有很多很多，历法与二十四节气的计算、华容道、井字棋、黑白棋、五子棋以及俄罗斯方块……你会惊讶地发现，再简单不过的事情背后，都藏着算法的神奇背影。不妨将本书放在案头慢慢品读，你将能看到算法如何深入我们的日常生活，如何重塑我们的世界。

你准备好了吗？接下来，这个世界算法将接管一切。

啊哈磊

《啊哈！算法》作者

致　　谢

本书的示例和思考来源于我多年的资料收集和面试题目，旨在通过现实生活中的有趣实例揭示算法的作用。本书来源于我博客中的算法专栏，在写作的过程中，很多人给予了我无私的帮助，在此我要向所有帮助过我的人致以诚挚的感谢。

首先，感谢我的家人给予的无条件的支持，没有他们的理解和鼓励，本书将无法按时完稿。

其次，感谢图灵的各位编辑老师在本书策划和编写过程中给予的指导和帮助，感谢本书的排版老师让书中的图表更加清晰和规范，感谢封面设计师潘建永和书签设计师Sneezry，你们非凡的创意和优秀的设计让这本书锦上添花。感谢王益、黄鑫和纪磊对本书的认同和推荐。

最后，我要感谢本书参考资料的所有作者，我已经尽力寻找所有资料的引用根源，但是仍有可能漏掉一些内容，对于没有提到的名字的作者，我感到十分抱歉，但是仍然要感谢你们。

前　　言

程序员与算法，这是一个永恒的话题，无论在哪个论坛，只要出现此类主题的帖子，一定会看到两种针锋相对的观点的“激烈碰撞”。其实泡过论坛的人都知道，两种观点“激烈辩论”的惨烈程度往往可以上升到互相问候先人的高度，即使是技术论坛也不例外。在准备此书之前，我在博客的“算法系列”专栏已经陆陆续续地写了有一年多的时间，在此期间，不断有读者问我：“程序员必须会算法吗？”我实在不想让我的博客成为喷满各种口水的是非之地，所以一般不正面回答，只是笼统地说些“各行各业情况都不尽相同”之类的话，避免站队。

程序员对算法通常怀有复杂的感情，算法很重要是大家的共识，但是是否每个程序员都必须学算法是主要的分歧点。本书是想重新定义程序员对算法的理解，并不想通过说教的方式给出到底是学还是不学的结论。很多人可能觉得像人工智能、视频与音频处理以及数据搜索与挖掘这样高大上的内容才能称为算法，往往觉得算法深不可测。但是这些其实都不是具体的算法，而是一系列算法的集合，这里面既有各种大名鼎鼎的算法，比如神经网络、遗传算法、离散傅里叶变换算法以及各种插值算法，也有不起眼的排序和概率计算的算法。你必须深入地了解它们，才会领略到算法的实质——解决问题。忽视这一点，片面地或抽象地理解算法，就会使对算法的理解变得形而上学。在我的博客里就有人留言质疑：“穷举也算是算法？”且不说搜索和枚举是算法的基础设计模式之一，单就那么多的NPC问题（比如著名的汉密尔顿回路问题，至今还没有找到多项式时间的算法），实际上，从只有穷举算法和其他随机搜索算法才能求解这一点看，任何人都不能小看它。

狭隘的算法定义会将自己局限在一个小角落里，从而错过了整个色彩缤纷的算法世界。本书将带你开启一段算法之旅，在这里，你将会看到各种构造算法的基础方法，比如贪婪法、分治法、动态规划法，等等，也可以通过一个个示例看到如何应用这些算法来解决实际问题。通过对“爱因斯坦的思考题”“三个水桶等分水”“妖怪与和尚过河问题”等趣味智力题的计算机求解算法设计，你可以领会到算法设计的三个关键问题，以及对这些问题的处理方法，为以后解决这样的问题提供举一反三的基础。

在生活中，凡是有乐趣的地方就有算法。本书将介绍生活中无处不在的算法。在历法计算的章节里，你会看到霍纳法则（Horner’s rule）的使用和求解一元高次方程的牛顿迭代法；音频播放器上跳动的频谱，背后是离散傅里叶变换算法；DOS时代著名的PCX图像文件格式使用的RLE压缩算法是如此简单，但是却非常有效；RSA加密算法的光环之下是朴实的欧几里得算法、蒙哥马利算法和米勒–拉宾算法；华容道游戏求解的简单穷举算法中还蕴藏着对棋盘状态的哈希算法……遗传算法神秘不可测，但是用遗传算法求解0-1背包问题只用了60多行代码。事实上，抛

开对遗传算法的深层次研究和在各种专业领域内的扩展应用，单就算法原理来说，它就是这么简单。深蓝战胜卡斯特罗之后，人类棋手在与计算机的博弈中就完全处于下风，人工智能真的这么神奇？人工智能确实是个神奇的领域，但就计算机下棋这件事来说，却并不怎么神奇，算法的基本原理简单得让人难以置信，看看第23章你就知道了。

算法之大，大到可以囊括宇宙万物的运行规律，算法之小，小到寥寥数行代码即可展现一个神奇的功能。算法是琐碎的，以至于常常被人们忽视，然而忽视算法能力的培养所带来的代价是巨大的，第1章介绍的环形队列的例子就是一个最好的说明。我面试过很多求职者，我常常会让他们手写一个算法，我的题目是这样的：有一个由若干正整数组成的数列，数列中的每个数都不超过32，已知数列中存在重复的数字，请给出一个算法找出这个数列中所有重复出现的数。我期望求职者给我一个正确的算法实现，接下来我会问这个算法的时间复杂度是什么，有没有考虑过存在一个$O(n)$时间复杂度的算法。大部分求职者都知道自己的算法是$O(n^2)$时间复杂度，但是都否认存在$O(n)$时间复杂度的算法。事实上这个题目是可以有$O(n)$时间复杂度的算法的，因为大家都忽略了一个重要的条件。这个题目并不难，但是仍有将近三分之一的面试者无法给出正确的算法，有的甚至还给我一张白纸。有人犯错误是正常现象，但是让我意外的是居然有三分之一的人写不出这个算法，算法设计的基本功被无视到这种地步是不正常的。

程序员谈到算法言必称一些高大上的词汇，但是这些专有名词大部分人是用不到的，以至于人们常常认为算法不过如此，不会又如何？这种思想变得极端就会让人忽视算法的基础设计能力，这才是最要命的。在我们维护的网络设备上，用户的数据关系错综复杂，一个对线性表进行二重循环都想不到的人又怎么可能会维护这些数据？我希望程序员们提高基础的算法能力，先从培养兴趣开始或许是一个不错的切入点。

本书挑选的算法例子，都围绕着“趣”字展开，都是简单且在生活中常见的算法，可能有些是你还没有意识到的。我上学的时候曾经做过一个MP3播放器程序，你可能觉得这主要就是利用一些音频解码算法吧？是的，这个是主要部分，但是一个功能完整的播放程序还用了很多你想不到的算法：为增加频谱显示和均衡器功能，使用了离散傅里叶变换算法；为计算频率功率谱，使用了加权平均值算法；为了匹配硬件输出设备与解码算法的性能差异，需要一个有多个缓冲区的队列管理音频数据块，这就引入了滑动窗口算法；为提供按照专辑名称或作者名称排序功能，使用了快速排序算法；为了平滑均衡器调节对音频的影响，使用了三次样条曲线插值算法；为了在两首歌曲之间切换时压制刺耳的杂音（通过填充一些舒适噪声的方式实现），还使用了正弦信号发生器算法。这些你都没有想到吧？其实还有更多的例子，比如大型项目管理软件中的工作节点排序功能和关键路径功能，背后支撑它们的却是简单的有向图拓扑排序算法。这是不是很有趣？生活中处处都是算法，程序员又怎么可能与算法绝缘？

再次重申一点，本书没有任何关于算法重要性的说教，当你看到本书时，我希望你的表情是“啊哈，原来如此！”，或者是“嗯，有意思！”，并从中获得乐趣。本书几乎所有章节都有相关算法实现和功能演示的代码，读者可以到我的博客（http://blog.csdn.net/orbit/）中下载，也可以到图灵社区本书主页（www.iTuring.cn/book/1605）下载使用。

目　　录

第 1 章 程序员与算法

本章的标题既然是“程序员与算法”，就必然要涉及一个基本问题，那就是“程序员是否必须会算法”。这是一个充满争议的问题，虽然并不像“生存还是毁灭”之类的选择那样艰难而沉重，但也绝不是一个轻松的话题。朋友们在我的“算法系列”博客专栏上发表的评论和回复，并不都是我所期待的赞美和鼓励，也常常会有一些冷言冷语。比如，“穷举也算是算法吗”或者“请你说明一下算法在 XX 系统中能起到什么作用”。

有一次，一个网友通过邮件问我：“你写的都是小儿科的东西，几十行代码就能搞定，能不能整一点高深的算法？”我反问他什么是他所理解的高深的算法，他答复说：“像遗传算法、蚁群算法之类的。”于是我给了他一个遗传算法求解 0-1 背包问题的例子（参见第 16 章），并告诉他，这也就是几十行代码的算法，怎么理解成是高深的算法？他刚开始不承认这是遗传算法，直到我给了他 Denis Cormier 公开在北卡罗来纳州立大学服务器上的遗传算法的源代码后，他才相信他一直认为深不可测的遗传算法的原理原来是这么简单。

还有一个网友直言我写的“用三个水桶等分 8 升水”之类的问题根本就称不上算法，他认为像“深蓝”那样的人工智能才算是算法。我告诉他计算机下棋的基本理论就是博弈树，或者再加一个专家系统。但是他认为博弈树也是很高深的算法，于是我给了他一个井字棋游戏（参见第 23 章），并告诉他，这就是博弈树搜索算法，非常智能，你绝对战胜不了它（因为井字棋游戏很简单，这个算法会把所有的状态都搜索完）。我相信他一定很震惊，因为这个算法也不超过 100 行代码。

对于上面提到的例子，我觉得主要原因在于大家对算法的理解有差异，很多人对算法的理解太片面，很多人觉得只有名字里包含“XX 算法”之类的东西才是算法。而我认为算法的本质是解决问题，只要是能解决问题的代码就是算法。在讨论程序员与算法这个问题之前，我们先探讨一个最基本的问题：什么是算法。

1.1　什么是算法

《算法导论》一书将算法（algorithm）描述为定义良好的计算过程，它取一个或一组值作为输入，并产生一个或一组值作为输出。Knuth 在《计算机程序设计艺术》一书中将算法描述为从一个步骤开始，按照既定的顺序执行完所有的步骤，最终结束（得到结果）的一个过程。Weiss 在《数据结构与算法分析》一书中将算法描述为一系列的计算步骤，将输入数据转换成输出的结果。

虽然没有被普遍接受的“算法”的正式定义，但是各种著作中对算法的基本要素或基本特征的定义都是明确的，Knuth 总结了算法的四大特征。

- 确定性。算法的每个步骤都是明确的，对结果的预期也是确定的。
- 有穷性。算法必须是由有限个步骤组成的过程，步骤的数量可能是几个，也可能是几百万个，但是必须有一个确定的结束条件。
- 可行性。一般来说，我们期望算法最后得出的是正确的结果，这意味着算法中的每一个步骤都是可行的。只要有一个步骤不可行，算法就是失败的，或者不能被称为某种算法。
- 输入和输出。算法总是要解决特定的问题，问题来源就是算法的输入，期望的结果就是算法的输出。没有输入的算法是没有意义的，没有输出的算法是没有用的。

算法需要一定的数学基础，但是没有任何文献资料将算法限定于只解决数学问题。有些人将贪婪法、分治法、动态规划法、线性规划法、搜索和枚举（包括穷尽枚举）等方法理解为算法，其实这些只是设计算法常用的设计模式（Knuth 称之为设计范式）。同样，计算机程序只是算法的一种存在形式，伪代码、流程图、各种符号和控制表格也是常见的算法展示形式。而顺序执行、并行执行（包括分布式计算）、递归方法和迭代方法则是常用的算法实现方法。

综合以上分析和引述，本人将算法定义为：*算法是为解决一个特定的问题而精心设计的一套数学模型以及在这套数学模型上的一系列操作步骤，这些操作步骤将问题描述的输入数据逐步处理、转换，并最后得到一个确定的结果。*使用“精心设计”一词，是因为我将算法的设计过程理解为人类头脑中知识、经验激烈碰撞的过程，将算法理解为最终“小宇宙爆发”一般得到的智力结果。

1.2　程序员必须要会算法吗

很多人可能是好莱坞大片看多了，以为计算机神通广大，但事实不是这样的。计算机其实是一种很傻的工具，傻到几乎没有智商（至少目前是这样）。它可以连续几年做同一件事情而毫无怨言，但是如果你不告诉它怎么做，它什么事情也不会做。最有创造性的活动其实是由一种被称为“程序员”的人做的，计算机做的只不过是人类不愿意做的体力活而已。比如图像识别技术，需要一个字节一个字节地处理数据，提取数据的特征值，然后在海量的数据中比较、匹配这些特征值，直到累得两眼昏花，人类才不会干这种傻事儿呢。计算机愿意做，但前提是你要告诉它怎

么做。算法可以理解为这样一种技术，它将告诉计算机怎么做。有人将编程理解为搭积木，直接用别人开发好的组件、库，甚至是类或 API 就行了，并且美其名曰“不用重复发明轮子”。我认为这其实就是所谓的系统集成，如果一个程序员每天的工作就是搭积木，那将是令人十分羡慕的事情，但是我知道，事实并不是这样的。这样搭积木式的编程计算机就可以做，没有必要让人来做，因为人工的成本高于计算机。我遇到的更多的是在论坛里发帖求助的人，比如“求代码，把一个固定格式的文本文件读入内存”，再比如“谁能帮我把这个结构数组排排序啊，书上的例子都是整数数组排序”。他们是如此地无助，如果不是论坛对回帖有积分奖励的话，恐怕不会有人理他们。

我要说的是，大多数程序员并不需要知道各种专业领域里的算法，但是你要会设计能够解决你面临问题的算法。一些领域内的经典问题，在前人的努力之下都有了高效的算法实现，本书的很多章节都介绍了这样的算法，比如稳定匹配问题、A*算法等。但是更多情况下，你所面临的问题并没有现成的算法实现，需要程序员具有创新的精神。算法设计需要具备很好的数学基础，但数学并不是唯一需要的知识，计算机技术的一些基础学科（比如数据结构）也是必需的知识，有人说：程序 = 算法+数据结构，这个虽然不完全正确，但是提到了计算机程序最重要的两点，那就是算法和数据结构。算法和数据结构永远是紧密联系在一起的，算法可以理解为解决问题的思想，这是程序中最具有创造性的部分，也是一个程序有别于另一个程序的关键点，而数据结构就是这种思想的载体。

再次重申一遍，我和大多数人一样，并不是要求每个程序员都精通各种算法。大多数程序员可能在整个职业生涯中都不会遇到像 ACM（Association for Computing Machinery）组织的国际大学生程序设计竞赛中的问题，但是说用不到数据结构和算法则是不可想象的。说数据结构和算法没用的人是因为他们用不到，用不到的原因是他们想不到，而想不到的原因是他们不会。请息怒，我不是要打击任何人，很多情况下确实是因为不会，所以才用不到，下面就是一个典型的例子。

1.2.1 一个队列引发的惨案

我所在的团队负责一款光接入网产品的“EPON 业务管理模块”的开发和维护工作，这是电信级的网络设备，因此对各方面性能的要求都非常高。有一天，一个负责集成测试的小伙儿跑过来对我说，今天的每日构造版本出现异常，所有线卡（承载数据业务的板卡）的上线时间比昨天的版本慢了 4 分钟左右。我很惊讶，对于一个电信级网络设备来说，每次加电后的线卡上线时间就是业务恢复时间，业务恢复时间这么慢是不能接受的。于是我检查了一下前一天的代码入库记录，很快就找到了问题所在。原来当前版本的任务列表中有这样一项功能，那就是记录线卡的数据变更日志，需求的描述是在线卡上维护一个日志缓冲区，每当有用户操作造成数据变更时，就记录一条变更信息，线卡上线时的批量数据同步也属于操作数据变更，也要计入日志。因为是嵌入式设备，线卡上日志缓冲区的大小受限制，最多只能存储 1000 条记录，当记录的日志超过 1000 条时，新增的日志记录将覆盖旧的记录，也就是说，这个日志缓冲区只保留最近写入的 1000 条记录。一个新来的小伙儿接受了这个任务，并在前一天下班前将代码签入库中（程序员要记住啊，

一定不要在临下班前签入代码）。他的实现方案大致是这样的（注释是我加上的）：

```
#define SYNC_LOG_CNT              1000
#define SYNC_LOG_MEMOVER_CNT      50

typedef struct
{
    INT32U logCnt;
    EPON_SYNC_LOG_DATA syncLogs[SYNC_LOG_CNT];
}EPON_SYNC_LOG;

EPON_SYNC_LOG s_EponSyncLog;

void Epon_Sync_Log_Add(EPON_SYNC_LOG_DATA*pLogData)
{
    INT32U i = 0;
    INT32U syncLogCnt = 0;

    syncLogCnt = s_EponSyncLog.logCnt;
    if(syncLogCnt>=SYNC_LOG_CNT)
    {
        /*缓冲区已满，向前移动 950 条记录，为新纪录腾出 50 条记录的空间*/
        memmove(s_EponSyncLog.syncLogs,
                s_EponSyncLog.syncLogs + SYNC_LOG_MEMOVER_CNT,
                (SYNC_LOG_CNT-SYNC_LOG_MEMOVER_CNT) * sizeof(EPON_SYNC_LOG_DATA));
        /*清空新腾出来的空间*/
        memset(s_EponSyncLog.syncLogs + (SYNC_LOG_CNT - SYNC_LOG_MEMOVER_CNT),
               0, SYNC_LOG_MEMOVER_CNT * sizeof(EPON_SYNC_LOG_DATA));
        /*写入当前一条日志*/
        memmove(s_EponSyncLog.syncLogs + (SYNC_LOG_CNT - SYNC_LOG_MEMOVER_CNT),
                pLogData, sizeof(EPON_SYNC_LOG_DATA));
        s_EponSyncLog.logCnt = SYNC_LOG_CNT - SYNC_LOG_MEMOVER_CNT + 1;

        return;
    }
        /*如果缓冲区有空间，则直接写入当前一条记录*/
    memmove(s_EponSyncLog.syncLogs + syncLogCnt,
            pLogData, sizeof(EPON_SYNC_LOG_DATA));
    s_EponSyncLog.logCnt++;
}
```

这个方案使用一个长度为 1000 条记录的数组存储日志，用一个计数器记录当前写入的有效日志条数，数据结构的设计中规中矩，但是当缓冲区满，需要覆盖旧记录时遇到了麻烦，因为每次都要移动数组中的前 999 条记录，才能为新记录腾出空间，这将使 Epon_Sync_Log_Add()函数的性能急剧恶化。考虑到这一点，小伙儿为他的方案设计了一个阈值，就是 SYNC_LOG_MEMOVER_CNT 常量定义的 50。当缓冲区满的时候，就一次性向前移动 950 条记录，腾出 50 条记录的空间，避免了每新增一条记录就要移动全部数据的情况。可见这个小伙儿还是动了一番脑子的，在 Epon_Sync_Log_Add()函数调用不是很频繁的情况下，在功能和性能之间做了个折中，根据自测的情况，他觉得还可以，于是就在下班前匆匆签入代码，没有来得及安排代码走查和同行评审。但是他没有考虑到线卡上线时需要批量同步数据的情况，在这种情况下，Epon_Sync_Log_Add()函数

被调用的频度仍然超出了这个阈值所能容忍的程度。通过对任务的性能进行分析，我们发现大量的时间都花费在 Epon_Sync_Log_Add() 函数中移动记录的操作上，即便是设计了阈值 SYNC_LOG_MEMOVER_CNT，性能依然很差。

其实，类似这样的固定长度缓冲区的读写，环形队列通常是最好的选择。下面我们来看一下环形队列的示意图，如图 1-1 所示。

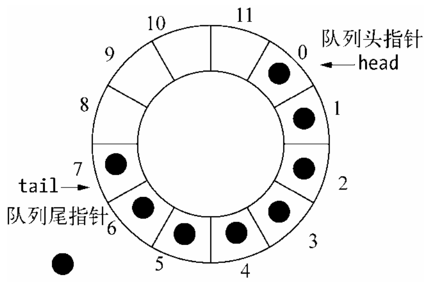


图 1-1 环形队列示意图

计算机内存中没有环形结构，因此环形队列都是用线性表来实现的，当数据指针到达线性表的尾部时，就将它转到 0 位置重新开始。实际编程的时候，也不需要每次都判断数据指针是否到达线性表的尾部，通常用取模运算对此做一致性处理。设模拟环形队列的线性表的长度是 N，队头指针为 head，队尾指针为 tail，则每增加一条记录，就可用以下方法计算新的队尾指针：

```
tail = (tail + 1) % N
```

对于本例的功能需求，当 tail + 1 等于 head 的时候，说明队列已满，此时只需将 head 指针向前移动一位，就可以在 tail 位置写入新的记录。使用环形队列，可以避免移动记录操作，本节开始时提到的性能问题就迎刃而解了。在这里，套用一句广告词：“没有做不到，只有想不到。”看看，我没说错吧？

1.2.2 我的第一个算法

我的第一份工作是为一个光栅图像矢量化软件编写一个图像预处理系统，这套光栅图像矢量化软件能够将从纸质工程图纸扫描得到的位图图纸识别成能被各种 CAD 软件处理的矢量化图形文件。在预处理系统中有一个功能是对已经二值化的光栅位图（黑白两色位图）进行污点消除。光栅位图上的污点可能是原始图纸上扫描前就存在的墨点，也可能是扫描仪引入的噪点，这些污点会对矢量化识别过程产生影响，会识别出错误的图形和符号，因此需要预先消除这些污点。

当时我不知道有小波算法，也不知道还有各种图像滤波算法，只是根据对问题的认识，给出了我的解决方案。首先我观察图纸文件，像直线、圆和弧线这样有意义的图形都是最少有 5 个点相互连在一起构成的，而污点一般都不会超过 5 个点连在一起（较大的污点都用其他的方法除掉了）。因此我给出了污点的定义：如果一个点周围与之相连的点的总数小于 5，则这几个相连在一起的点就是一个污点。根据这个定义，我给出了我的算法：从位图的第一个点开始搜索，如果这个点是 1（1 表示黑色，是图纸上的点；0 表示白色，是图纸背景颜色），就将相连点计数器加 1，然后继续向这个点相连的 8 个方向分别搜索，如果某个方向上的相邻点是 0 就停止这个方向的搜索。如果搜索到的相连点超过 4 个，说明这个点是某个图形上的点，就退出这个点的搜索。如果搜索完成后得到的相连的点小于或等于 4 个，就说明这个点是一个污点，需要将其颜色置为

0（清除污点）。

算法实现首先定义搜索过程中存储相连点信息的数据结构，这个数据结构定义如下：

```
typedef struct tagRESULT
{
    POINT pts[MAX_DIRTY_POINT];/*记录搜索过的前 5 个点的位置*/
    int count;
}RESULT;
```

这个数据结构有两个属性，count是搜索过程中发现的相连点的个数，pts是记录这些相连点位置的线性表。记录这些点的位置是为了在搜索结束后，如果判定这些点是污点，可以利用这些记录的位置信息直接清除这些点的颜色。

```
/*8 个方向*/
POINT dir[] = { {-1, 0}, {-1, -1}, {0, -1}, {1, -1}, {1, 0}, {1, 1}, {0, 1}, {-1, 1} };

void SearchDirty(char bmp[MAX_BMP_WIDTH][MAX_BMP_HEIGHT]
                 int x, int y, RESULT *result)
{
    for(int i = 0; i < sizeof(dir)/sizeof(dir[0]); i++)
    {
        int nx = x + dir[i].x;
        int ny = y + dir[i].y;
        if( (nx >= 0 && nx < MAX_BMP_WIDTH)
            && (ny >= 0 && ny < MAX_BMP_HEIGHT)
            && (bmp[nx][ny] == 1) )
        {
            if(result->count < MAX_DIRTY_POINT)
            {
                /*记录前 MAX_DIRTY_POINT 个点的位置*/
                result->pts[result->count].x = nx;
                result->pts[result->count].y = ny;
            }
            result->count++;
            if(result->count > MAX_DIRTY_POINT)
                break;

            SearchDirty(bmp, nx, ny, result);
        }
    }
}
```

向 8 个方向搜索使用了预置的矢量数组 dir，这是迷宫或棋盘类游戏搜索惯用的模式，本书介绍的算法会多次使用这种模式。SearchDirty()函数递归地调用自己，实现对 8 个方向的连通性搜索，最后的结果存在 result 中，如果 count 的个数大于 4，说明[x, y]位置的点是正常图形上的点，如果 count 的个数小于或等于 4，则说明[x, y]位置相邻的这个点是一个污点。污点相邻的点的位置都被记录在 pts 中，将这些位置的位图数据置 0 就消除了污点。算法没有做任何优化，不过好在图纸上大部分都是白色背景，需要搜索的点并不多。打开测试图纸一试，速度并不慢，效果也很好，几个故意点上去做测试用的污点都没有了，小的噪点也没有了，图纸一下就变白了。

不过这段代码最终并没有成为那个软件的一部分，学过机械制图的同学可能看出来了，这个算法会将一些细小的虚线和点划线一并干掉。

这是一个微不足道的问题，但却是我第一次为解决（当然，未遂）问题而设计了一个算法，并最终用程序将其实现。它让我领悟到了一个道理，软件被编写出来就是为了解决问题的，程序员的任务就是设计解决这些问题的算法。成功固然高兴，失败也没有什么代价，可以随时卷土重来。不要小看这些事情，不要以为只有各种专业领域的程序才会用到算法，每一个微小的设计都是算法创造性的体现，即使失败，也比放弃强。

1.3 算法的乐趣在哪里

算法有很多种存在形式，编写计算机程序只是其中一种，是程序员惯用的方式，本书要介绍的内容就是如何以计算机程序的方式研究算法。1.2 节介绍的两个例子都是我亲身经历过的事情，程序员在大部分时间里都是处理一些平凡而琐碎的程序，但有时候也需要做一些创造性的工作。记住，程序员就是计算机的“上帝”，计算机能解决问题是因为它的“上帝”告诉它怎么做。那么，当问题来临的时候，“上帝”是到各种论坛上发帖子求代码，还是自己解决问题？

如果要自己解决问题，应该如何解决问题？为什么要自己解决问题？先来回答第一个问题——如何设计算法解决问题？人类解决问题的方式是当遇到一个问题时，首先从大脑中搜索已有的知识和经验，寻找它们之间具有关联的地方，将一个未知问题做适当的转换，转化成一个或多个已知问题进行求解，最后综合起来得到原始问题的解决方案。编写计算机程序实现算法，让计算机帮我们解决问题的过程也不例外，也需要一定的知识和经验。为了让计算机帮我们解决问题，就要设计计算机能理解的算法程序。而设计算法程序的第一步就是要让计算机理解问题是什么。这就需要建立现实问题的数学模型。建模过程就是一个对现实问题的抽象过程，运用逻辑思维能力，抓住问题的主要因素，忽略次要因素。建立数学模型之后，第二个要考虑的问题就是输入输出问题，输入就是将自然语言或人类能够理解的其他表达方式描述的问题转换为数学模型中的数据，输出就是将数学模型中表达的运算结果转换成自然语言或人类能够理解的其他表达方式。最后就是算法的设计，其实就是设计一套对数学模型中的数据的操作和转换步骤，使其能演化出最终的结果。

数学模型、输入输出方法和算法步骤是编写计算机算法程序的三大关键因素。对于非常复杂的问题，建立数学模型是非常难的事情，比如天文物理学家研究的“宇宙大爆炸”模型，再比如热力学研究的复杂几何体冷却模型，等等。不过，这不是本书探讨的范围，程序员遇到的问题更多的不是这种复杂的理论问题，而是软件开发过程中常用和常见的问题，这些问题简单，但并不枯燥乏味。对于简单的计算机算法而言，建立数学模型实际上就是设计合适的数据结构的问题。这又引出了前面提到的话题，数据结构在算法设计过程中扮演着非常重要的角色。输入输出方式和算法步骤设计都是基于相应的数据结构设计的，相应的数据结构要能很方便地将原始问题转换成数据结构中的各个属性，也要能很方便地将数据结构中的结果以人们能够理解的方式输出，同

时，也要为算法转换过程中各个步骤的演化提供最便利的支持。使用线性表还是关联结构，使用树还是图，都是在设计输入输出和算法步骤时就要考虑的问题。

为什么要自己解决问题？爱因斯坦说过："兴趣是最好的老师。"这就是说，只要一个人对某事物产生兴趣，就会主动去学习、去研究，并在学习和研究的过程中产生愉快的情绪。我把从算法中体会到的乐趣分成三个层次：初级层次是找到特定的算法解决特定的实际问题，这种乐趣是解决问题后的成就感；中级层次是有些算法本身就是充满乐趣的，搞明白这种算法的原理并写出算法的程序代码，能为自己今后的工作带来便利；高级层次是自己设计算法解决问题，让其他人可以利用你的算法享受到初级层次的乐趣。有时候问题可能是别人没有遇到过的，没有已知的解法，这种情况下只能自己解决问题。这是本书一直强调算法的乐趣的原因。只有体会到乐趣，才有动力去学习和研究，而这种学习和研究的结果是为自己带来正向的激励，为今后的工作带来便利。回想一下 1.2.1 节的例子，环形队列相关的算法是固定长度缓冲区读写的常用模式，如果知道这一点，就不会有这种问题了。

1.4 算法与代码

本书讲到的算法都是以计算机程序作为载体展示的，其基本形式就是程序代码。作为一个软件开发人员，你希望看到什么样的代码？是这样的代码：

```
double kg = gScale * 102.1 + 55.3;
NotifyModule1(kk);
double kl1 = kg / l_mask;
NotifyModule2(kl1);
double kl2 = kg * 1.25 / l_mask;
NotifyModule2(kl2);
```

还是这样的代码：

```
double globalKerp = GetGlobalKerp();
NotifyGlobalModule(globalKerp);
double localKrep = globalKerp / localMask;
NotifyLocalModule(localKrep);
double localKrepBoost = globalKerp * 1.25 / localMask;
NotifyLocalModule(localKrepBoost);
```

程序员都有一种直觉，那就是能看懂的代码就是好代码。但是"能看懂"是一个非常主观的感觉，同样的代码给不同的人看，能否看懂有着天壤之别。《重构》一书的作者为不好的代码总结了 21 条"坏味道"规律，希望能够对号入座地判断一下代码中的"坏代码"。但是这 21 条规律仍然太主观，于是人们又给代码制定了很多量化指标，比如代码注释率（这个指标因为没有意义，已经被很多组织抛弃了）、平均源代码文件长度、平均函数长度、平均代码依赖度、代码嵌套深度、测试用例覆盖度，等等。做这些工作的目的在于人们希望看到漂亮的代码，这不仅仅是主观审美的需要，更是客观上对软件质量的不懈追求。漂亮的代码有助于改善软件的质量，这已经是公认的事实，因为程序员在把他们的代码变得漂亮的过程中，能够通过一些细小却又重要的方式改善代码的质量，这些细小却又重要的方式包括但不限于更好的设计、可测试性和可维护性

等方面的方法。

在保证软件行为正确性的基础上，人们都用什么词来形容好的代码呢？好看、漂亮、整洁、优雅、艺术品、像诗一样？我看过很多软件的代码，有开源软件的代码，也有商业软件的代码，好的代码给我的感觉就是以上这些形容词，当然也见过不好的代码，给我的感觉就是“一堆代码”而已。我在写“算法系列”博客专栏的时候，就特别注意这一点，即便别人已经发布过类似的算法实现，我也希望我的算法呈现出来的是完全不一样的代码。设计算法也和设计软件一样，应该是漂亮的代码，如果几百行代码堆在一起，不分主次，关系凌乱，只是最后堆出了一个正确的结果，这不是我所希望的代码，既虐人又虐己。大部分人来看你的博客，应该还是为了看懂吧。在我准备这本书的时候，我把很多算法又重新写了一遍，不仅算法有趣，研究代码也是一种乐趣。如果算法本身很有趣，但是最后的代码实现却是毫无美感的“一堆代码”，那真是太扫兴了。

1.5 总结

本章借用了多部知名著作中对算法的定义，只是想让大家对算法有一个“宽容”一点的理解。通过我亲身经历的两个例子，说明了程序员与算法之间“剪不断，理还乱”的关系。除此之外，还简单探讨了算法乐趣的来源、算法和代码的关系，以及研究代码本身的乐趣等内容。

如果你认同我的观点，就可以继续阅读本书了。本书的每一章都是独立的，没有前后关系，你可以根据自己的喜好直接阅读相关的章节。希望本书能使你有所收获，并体会到算法的乐趣。

1.6 参考资料

[1] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[2] Knuth D E. *The Art of Computer Programming* (*Third Edition*), Vol 1. Addison-Wesley, 1997

[3] Weiss M A. *Data Structures and Algorithm Analysis* (*Second Edition*). Addison-Wesley, 2001

[4] Oram A, Wilson G. *Beautiful Code*. O'Reilly Media, Inc, 2007

[5] Fowler M, et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999

第 2 章 算法设计的基础

看到这里，说明你对算法设计感兴趣。编写程序开发软件是冒险者的游戏，需要胆大心细，设计一个解决实际问题的算法尤其如此。现实问题复杂多样，对应的算法也是复杂多样，形态各异，但是这些算法都遵循一些特定的方法和模式。就算法模式而言，处理各种求最优解问题时，人们常用贪婪法、动态规划法等算法模式；处理迷宫类问题时，穷尽式的枚举和回溯是常用的模式。就算法的实现方法而言，如果算法需要频繁地查表操作，那么数据结构的设计通常会选择有序表来实现；反过来，当设计的算法用到了树和图这样的数据结构时，含有递归结构的方法就常常伴随它们左右。

算法设计是个复杂的内容，单就这个话题就可以写一本书了。克林伯格的《算法设计》就是一本这样的书，掌握了算法设计的基本内容之后，就可以去啃《算法导论》或其他各种竞赛类算法的书了，但这都不是本书的重点。本书的目的是展示算法的有趣之处，希望通过一些简单有趣的算法改变程序员对算法的固有印象，我们不去研究那些各个行业领域内的复杂算法，只来讨论一些通用的、共性的东西。

2.1 程序的基本结构

从大的方面来考量算法问题，相对于并行算法，本书介绍的都是串行算法的设计方法。按照冯·诺依曼计算机体系的设计，计算机的 CPU 每次只能串行执行一条指令，即使那些号称支持多线程的操作系统，其实际效果也是“宏观上并行，微观上串行”。顺序执行、循环和分支跳转是程序设计的三大基本结构，算法也是程序，千姿百态的算法也是由这三大基础结构构成的。

2.1.1 顺序执行

顺序执行是算法的基础结构，循环体结构的每个循环体内也是顺序执行的，分支和跳转的每个分支内也是顺序执行的。假如算法中某个操作需要几个步骤完成，每个步骤都依赖于前一个步骤，将前一个步骤的输出作为下一个步骤的输入，中间不能打断和调整顺序，这样的结构就是算

法的顺序执行结构（如图 2-1 所示）。



图 2-1 顺序执行结构图

以雇员工资打印算法为例，必须先统计出雇员的出勤情况，然后才能计算工资，最后才能打印工资单，这三个步骤顺序不能打乱，否则将无法得到正确的结果。第 11 章中计算太阳的地心视黄经算法，需要计算出平黄经，然后修正地心章动，修正交角章动，修正光行差，最后得到地心视黄经。其中修正地心章动、修正交角章动和修正光行差这三个步骤是对平黄经值的修正，没有前后关系，可以互换顺序，虽然算法实现是顺序方式，但是它们不是严格的顺序执行结构。

顺序执行结构虽然简单，但是却具有重要的意义。算法的基本特征之一就是确定性，因此算法里的顺序结构必须是明确的，其结果是不变的。除了确定性，顺序执行结构还具有封闭性特征，也就是说，无论上一步的结果如何，都会继续执行下一步，不受外部条件和内部因素的影响。

2.1.2 循环结构

循环结构也是算法中一种很重要的控制流程，循环被定义为在算法中只出现一次但是却有可能被执行多次的一段逻辑体。从实际应用角度看，稍微复杂一点的算法都会用到循环结构。循环结构一般由三部分组成：循环初始化、循环体和循环条件（退出条件）。循环初始化部分一般做些循环体控制状态的初始化工作，包括循环条件的初始化。循环体可以是顺序执行结构，也可以带有分支跳转结构，甚至可以是个循环体（多重循环结构）。循环条件可以是简单的计数器，也可以是复杂的逻辑判断，它定义循环的执行条件或退出条件。有少数编程语言提供一些特殊的指令，可以控制循环结构的流程和退出，比如 C 语言的 `continue` 和 `break` 语句，但是这种控制方式不具备算法通用性。

从数据结构方面看，涉及线性表的遍历和查找操作，一般都会用到循环结构，比如多项式求和算法和各种排序算法。维基百科的“算法”条目给出了一个求最大数的例子，其算法实现如下：

```
int max(int *values, int size)
{
    int mval = *values;
    int i;
    for(i = 1; i < size; i++)
    {
        if(values[i] > mval)
            mval = values[i];
    }

    return mval;
}
```

for 语句块的代码就是 C 语言形式的循环结构，循环条件是 i < size。

关于循环也有很多有趣的话题。举个例子，如果算法操作的数据结构是二维数组，通常都会用到两重循环，但是也可以用单循环遍历二维数组，第 19 章介绍数独游戏的解法的时候，对小九宫格的遍历就多次使用了这种技巧，比如初始化一个小九宫格的代码可能是这样的：

```
for(int i = 0; i < 9; i++)
{
    int row = i / 3;
    int col = i % 3;
    game->cells[row][col].fixed = false;
}
```

只要介绍循环，就不得不提*递归*，一些文献资料将递归作为一种独立的程序控制方法介绍，但是更多的资料将递归看作是循环的一种替代形式。这两种形式看起来差异很大，但是本质都是一样的，递归结构通常都可以用复杂一点的循环形式代替，特别是尾递归形式，可以直接替换成循环结构。递归结构一般由递归关系定义和递归终止条件两部分组成，递归关系定义就是对问题的分解，是指向递归终止条件转化的规则，而递归终止条件通常就是问题分解到最小规模时，这个最小规模的问题对应的结果。递归方法符合人类思考问题的方式，它可以使算法结构简单，过程简洁。对于树和图这样的数据结构，递归方法更是具有循环形式无法比拟的优势，下面给出了 FindTNode()函数递归方式实现的二叉树查找算法，大家可以体会一下递归的优美。

```
bool FindTNode(TNODE *tr, int key)
{
    if(tr == NULL)
        return false;
    if(tr->key == key)
        return true;

    if(key < tr->key)
        return FindTNode(tr->left, key);
    else
        return FindTNode(tr->right, key);
}
```

尾递归是尾调用[①]的一种特殊情况，也是递归结构的一种特殊形式。编译器一般都可以对尾递归进行优化（尾调用消除技术），直接利用当前函数的栈帧，将尾调用处理成循环的形式。实际上，即便不使用编译器的优化，尾递归也可以很容易转化成循环形式。前文给出的 FindTNode() 函数其实就是尾递归，可以很容易将其转化成循环形式，代码如下所示：

```
bool FindTNode(TNODE *tr, int key)
{
    TNODE *curNode = tr;
    while(curNode != NULL)
    {
        if(curNode->key == key)
```

① 尾调用是指一个函数里的最后一个动作是调用一个函数的情形，这个函数调用的返回值直接被当前函数作为返回值。

```
            return true;

        if(key < curNode->key)
            curNode = curNode->left;
        else
            curNode = curNode->right;
    }

    return false;
}
```

递归结构使用的函数递归调用，会增加任务的栈空间使用，用递归方法解决问题的规模受系统栈空间的约束，除此之外，函数调用时的参数入栈和出栈也会降低算法的效率。

2.1.3 分支和跳转结构

顺序结构可以解决计算、输入和输出等问题，但是不能作判断和选择。现实生活中的很多问题都需要进行判断和选择，处理这些问题，关键在于对条件的判断。分支和跳转结构在程序中扮演着重要的角色，正是由于有了分支和跳转，程序才能够产生多种多样的结果。算法设计也离不开分支和跳转结构，根据对条件的判断，选择合适的处理步骤，是算法实现过程中常用的逻辑。分支和跳转结构算法设计的关键是设计分支条件和算法的跳转流程，一般一个分支条件对应一个处理流程。算法在执行的过程中，根据构造的分支条件进行判断，根据判断的结果转入相应的处理流程继续执行。

根据跳转分支的个数，分支结构又可细分为单分支结构、双分支结构、嵌套分支结构和多分支结构（switch-case 结构）。单分支结构一般可表示为：

```
if(条件)
{
    分支流程
}
```

根据分支条件的判断结果，{}内的分支流程可能被执行，也可能不被执行。从算法构造的角度看，单分支结构多被用在根据条件判断，需要在正常处理流程中插入一些特殊处理的情况。比如，计算一年有多少天，如果是闰年就需要额外多加一天，就可以采用单分支结构，代码如下所示：

```
int days_per_year = 365;
if(((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0))
{
    days_per_year += 1;
}
return days_per_year;
```

双分支结构适合那种非“真”即“假”的流程处理，两个分支为互斥流程，执行一个分支就必然不会执行另一个分支。双分支结构一般可表示为：

```
if(条件)
{
    分支流程 1
```

```
}
else
{
    分支流程 2
}
```

分支流程 1 和分支流程 2 是根据条件互斥执行的。单分支结构有时候可以看作是双分支结构的一种特殊情况，即分支流程 2 是空的情况。

当某个条件分支的处理流程中又包含分支条件时，就构成嵌套分支结构，嵌套分支结构可以表示为：

```
if(条件 1)
{
    if(条件 2)
    {
        分支流程
    }
}
```

当算法的某个流程处理存在多级条件判断的时候，就会用上嵌套分支结构，但是过深的嵌套分支结构会使得算法代码条理不清楚，降低代码的可读性。一般嵌套分支结构不要超过三层，否则的话就要考虑调整算法，或者用函数封装替换分支代码，以提高算法代码的可读性。对于简单条件的多层嵌套，可以使用组合条件来避免多层分支嵌套，比如前面的两层嵌套结构就可以调整为：

```
if(条件 1 && 条件 2)
{
    分支流程
}
```

当算法的某个步骤需要多重筛选条件时，就会用到多分支结构，多分支结构可以表示为：

```
if(条件 1)
{
    分支流程 1
}
else if(条件 2)
{
    分支流程 2
}
...
else
{
    分支流程 n
}
```

和双分支结构一样，多分支结构的每个分支流程也是互斥执行的。生活中有很多情况都不是非真即假的两种选择，因此多分支结构在算法中也经常用到，比如给学生打评语的算法，就需要根据分数的区间将评语定位为“优秀”、“优良”、“良好”、“及格”，等等。有一些编程语言提供了类似于 switch-case 的结构，可以在某些情况下替换多分支结构。switch-case 结构的优点是对分支条件只进行一次判断就可以决定代码的分支流程，避免多次判断分支条件，但是缺点就是不

能进行复杂的条件判断，比如 C 语言的 switch-case 结构，其分支判断条件就只支持整数类型的常量表达式。

过长的多分支结构常被视为软件中的不良结构，因为它违背了 OCP 原则（开放、封闭原则），每当需要新增一种条件判断处理时，就要新增一个 if-else 分支。在很多情况下，使用函数表结构是避免过长的多分支结构的有效方法，下面就是“狼、羊和菜过河”问题的求解算法中用函数表结构代替过长的多分支结构的例子。求解“狼、羊和菜过河”问题，从一个状态过渡到下一个状态是由农夫的动作驱动的，农夫一共可以采取 8 种动作，每种动作都对应一个状态转变处理流程。如果采用 if-else 多分支结构，处理状态转换的代码将会非常长，为了避免过长的分支跳转代码，算法采用了函数表结构。首先声明函数表项的定义：

```
typedef bool (*ProcessNextFuncPtr)(const ItemState& current, ItemState& next);
struct ActionProcess
{
    Action             act;
    ProcessNextFuncPtr processFunc;
};
```

然后分别为农夫的 8 个动作指定处理函数，得到函数表的定义：

```
ActionProcess actMap[] =
{
    { FARMER_GO,                 ProcessFarmerGo                   },
    { FARMER_GO_TAKE_WOLF,       ProcessFarmerGoTakeWolf         },
    { FARMER_GO_TAKE_SHEEP,      ProcessFarmerGoTakeSheep        },
    { FARMER_GO_TAKE_VEGETABLE,  ProcessFarmerGoTakeVegetable   },
    { FARMER_BACK,               ProcessFarmerBack                 },
    { FARMER_BACK_TAKE_WOLF,     ProcessFarmerBackTakeWolf       },
    { FARMER_BACK_TAKE_SHEEP,    ProcessFarmerBackTakeSheep      },
    { FARMER_BACK_TAKE_VEGETABLE, ProcessFarmerBackTakeVegetable  }
};
```

如果用 if-else 结构，处理状态转换可能需要 30 多行代码，而利用这个函数表，处理状态转换的代码只有几行：

```
ItemState next;
for(int i = 0; i < sizeof(actMap)/sizeof(actMap[0]); i++)
{
    if(actMap[i].act == action)
    {
        actMap[i].processFunc(current, next);
        break;
    }
}
```

如果将这个函数表存入一个关联容器（比如 std::map）中，则循环体的代码都可以省去。如果随着算法演化，有新的动作需要处理，则只需要在函数表中添加新的条目即可，状态转换的代码不需要做任何改动。

算法需要确定性，分支和跳转看似使得算法具有不确定性，但是实际上，分支的判断和选择

都是在所有已确定处理流程的框架中进行的，也就是说，这些选择都是算法确定范围之内的选择，对算法确定性没有影响。虽然分支和跳转是算法构造的基础结构，但是如果能采用精心设计的一致性处理逻辑避免分支和跳转，通常算法会具有更好的结构。前面提到的函数表就是一个一致性处理的例子，第 1 章提到的环形队列的例子中，对尾指针移动的处理也是一个例子，如果不采用对 N 取模的一致性处理，则每次指针移动时都要做是否已经到表尾的判断处理。

2.2 算法实现与数据结构

计算机体系中的数据，是指能被计算机识别和处理的各种符号的总称。人类所能识别的各种数据，比如文字、语言和图像，在计算机内都是以二进制形式存在的，但是这些二进制数据之间存在着各种组织关系。我们通常说的数据结构，其实包含了两层意思，一是指相互之间存在某种特定关系的数据的集合，二是指数据之间的相互关系，也就是数据的逻辑结构。因此，当我们说定义数据结构时，除了定义数据之间的相互关系，还包括根据这些关系组织在一起的数据。在建立数学模型的阶段，我们说的数据结构更偏重于定义数据之间的相互关系，设计具体的算法步骤时，考虑的是如何对构建在这些数据关系之上的实际数据进行加工和处理。

算法和数据结构关系紧密，数据结构是算法设计的基础，不合适的数据结构设计，有可能导致无法设计算法的演算步骤，从而无法实现算法。数据之间常见的逻辑结构包括线性结构、关联结构（集合、映射）、树形结构和图形结构，也有一些资料将树形结构看作是图形结构的一种特殊形式，但是因为这两种数据结构在数据的组织和定义方式上存在很大的差异，更多的资料还是将它们分为两种结构。接下来我们讨论一下算法设计常用的几种基本数据结构，对于简单的问题，应用这些基本数据结构就可以解决，但是对于复杂的算法，往往需要将这些基本的数据结构组合起来形成更复杂的逻辑结构。

2.2.1 基本数据结构在算法设计中的应用

线性表是数据结构中最简单的基本数据结构。线性表的使用和维护都很简单，这一特点使其成为很多算法的基础。数组、链表、栈和队列是四种最常见的线性表，其外部行为和接口都各有特色，本节就简单介绍一下这四种基本数据结构的特点及其在算法设计中的应用。

1. 数组

数组（array）是一种相对比较简单的数据组织关系，所有数据元素存储在一片连续的区域内。对数组的访问方式一般是通过下标直接访问数组元素，除此之外，对数组的基本操作还有插入、删除和查找。数组元素的直接访问几乎没有开销，但是插入和删除操作需要移动数组元素，开销比较大，因此在插入和删除操作比较频繁的场合下，不适合使用数组。在数组中查找一个元素的时间复杂度是 $O(n)$，如果数组元素是有序存储的，则使用二分查找可以将时间复杂度降为 $O(\lg n)$。

在数组中存储的数组元素，除了数组元素的值需要关注之外，数组元素的下标也是一个很有用的属性，有时候可以巧妙地利用下标简化一些算法的实现方式。例如，有若干个数存放在 `value`

数组中，这些数的取值范围是[1-100]，请设计一个算法统计一下这些数中相同的数出现的次数。经分析发现，虽然 value 中数字的个数很多，但是范围并不大，可以设计一个有 100 个元素的数组，数组元素的下标对应数字，数组元素的值就是对应数字出现的次数，只需如下两行代码即可完成统计工作：

```
for(int i = 0; i < count; i++)
    numCount[values[i] - 1]++;
```

虽然对于那些没有出现过的数字也需要占用 numCount[]数组的一个位置，但是这点空间上的开销是可以接受的。

相对于固定长度的数组，还有可变长度的数组，比如 C++的 STL 提供的 std::vector，可变长数组具有数组的访问效率，同时长度可随着数据元素的增加而变长，使用场合比定长数组灵活。除了用一维数组表示线性表之外，还可以用多维数组表示更复杂的局面，比如棋盘类游戏可以使用二维数组表示棋盘状态，魔方类的游戏还可能用到三维数组，需要根据问题的情况灵活运用。

2. 链表

在线性表的长度不能确定的场合，一般会采用链表（linked list）的形式。链表结构的每个节点数据都由两个域组成，一个是存放实际数据元素的数据域，另一个就是构成链式结构的指针域。对于单向链表，指针域只有一个后向指针，对于双向链表，指针域由一个后向指针和一个前向指针组成。链表的插入和删除只需要修改指针域的指针指向即可完成，比数组的插入和删除操作效率高，但是访问数据元素的效率比较低，需要从链表头部向后（或向前）搜索，查找操作的时间复杂度是 $O(n)$。理论上链表的长度是不受限制的，实际使用链表时，常受存储器空间的限制，使得链表长度也不能无限增长，但是链表长度可动态变化这一点，比数组具有很大的优势。

单向链表只能在一个方向上遍历链表节点，从一个节点开始遍历到链表的尾部节点就停止。双向链表可以从两个方向遍历链表节点，从一个节点开始，向前遍历到链表头部节点停止，向后遍历到链表尾部节点停止。在某些应用场合，还可以将链表尾部节点的后向指针指向链表头部节点（对于双向链表，其头部节点的前向指针同时指向链表的尾部节点），构成一个环形链表。环形链表中头部节点和尾部节点的概念已经弱化，从任何一个节点开始都可以遍历整个链表。

链表的头节点作为整个链表遍历的起点，是一个比较特殊的节点，需要特殊处理，尤其是在插入和删除节点的时候。如果节点插入在头节点之前，或者删除的节点就是头节点，就需要调整链表头节点的指针，否则的话，仍用原来保存的头节点访问链表，就可能跳过新插入的节点，或操作已经失效的指针。为了解决这个问题，人们设计了一种在链表中使用固定头节点的方法，这个固定的头节点称为“表头节点”，也被称为“哑节点”（dummy node），《算法导论》一书将其称为“哨兵节点”。表头节点可以是一个没有数据域、只有指针域的特殊节点，也可以是和其他节点类型一样的节点（数据域不使用），更多的情况是在表头节点的数据域中放置一些与链表有关的状态信息，比如当前链表中的数据元素节点个数。表头节点的指针域始终指向第一个实际链表节点，如果表头节点的指针域是 NULL，则表示这个链表是空表。使用表头节点的好处有两个，

一个好处是无论链表是否为空表，始终有一个能标识链表的头节点，可以用一致的方法处理空链表和非空链表；另一个好处是对链表进行插入、删除和遍历操作时，不需要对数据元素的首节点和中间节点做差异处理，对每个节点的操作可以做到一致性。

除了查找和访问的效率没有数组高之外，链表的每个节点都要额外存储一个指针域，因此需要一定的存储开销。对于一些插入和删除操作比较少，查找、遍历操作比较多的场合，应该优先选择使用可变长数组代替链表。

3. 栈

栈（stack）是一种特殊的线性表，其特殊性在于只能在表的一端插入和删除数组元素，插入和删除动作分别被称为“入栈”和“出栈”。严格来说，栈不是一种数据存储方式，而是一种逻辑管理方式，它遵循“后进先出”（Last In First Out）的原则管理和维护表中的数据。栈的数据存储方式可以采用数组，也可以使用链表，分别被称为“顺序栈”和“链式栈”，但是无论采用何种存储方式，其外部行为都是一样的，即只能通过“出栈”和“入栈”的方式在数据表的一端操作数据。

栈是一种非常有用的数据结构，利用栈的一些特性，可以将某算法的递归实现转换成非递归实现，在使用穷尽搜索方法时，也会使用栈保存当前的状态，有时候，广度优先搜索和深度优先搜索的差异仅仅是使用栈还是使用队列。下面是一个判断表达式的括号是否匹配的小算法，可以体会一下这种“先进后出”的数据结构的特点。

```
bool IsMatchBrackets(const std::string& express)
{
    std::stack<std::string::value_type> epStk;
    std::string::size_type i;
    for(i = 0; i < express.length(); i++)
    {
        if(IsLeftBrackets(express[i]))
        {
            epStk.push(express[i]);
        }
        if(IsRightBrackets(express[i]))
        {
            if(epStk.empty())
                return false;
            epStk.pop();
        }
    }

    return epStk.empty();
}
```

4. 队列

队列（queue）也是一种特殊的线性表，普通的队列只能在表的一端插入数据，在另一端删除数据，不能在队列的其他位置插入和删除数据。插入和删除动作分别被称为“入队”和“出队”，

能执行“入队”的一端称为“后端”（rear），能执行出队的一端称为“前端”（front）。与栈一样，队列也不是一种数据存储方式，而是一种逻辑管理方式，它遵循“先进先出（First In First Out）”的原则管理和维护表中的数据。队列的数据存储方式可以采用数组，也可以使用链表。

队列有多种使用形式，比如第 1 章介绍过的环形队列（循环队列），还有可以在队列的两端都执行“入队”和“出队”操作的双端队列（double-ended queue），还有给每个数据元素打上优先级标签的优先级队列（priority queue），等等。队列在算法中的应用非常广泛，比如图的广度优先搜索算法，就使用一个队列存放与当前搜索节点有边相连的所有邻接点，以先进先出的原则一个一个地处理队列中的节点。操作系统中的线程调度算法，常使用带优先级的队列管理就绪线程列表，高优先级的线程插入在队列的前端，获得优先调度（出队）的机会。队列也是不同速率的 IO 设备之间缓冲区管理的常用方式，比如打印机打印速度比较慢，操作系统会为每个打印机维护一个打印队列，对不同进程提交的打印操作做入队管理，可以避免因一个大文件打印时间过长造成其他进程无法提交打印操作的问题。网络设备中也普遍使用队列来管理数据报文的发送和接收，以匹配不同速率的设备之间的数据传输。

2.2.2 复杂数据结构在算法设计中的应用

上一节讨论的基本数据结构都属于线性表范围，表中的数据元素之间没有关系，只是通过不同的组织和管理方式将每个数据元素维护在一个线性表中。本节将介绍的这些数据结构不是简单的线性表，并且每个数据元素之间也可能存在关系，比如树的节点之间存在父子关系，图的节点之间存在邻接关系，等等。之所以被称为“复杂数据结构”，是因为相关的插入、删除操作不仅对数据元素进行操作，还要同时维护数据元素之间的关系。

1. 树

树（tree）是一种表达数据之间层次关系的数据结构，树中的每个节点有 0 个或多个子节点，但是只有一个父节点，父节点为空的节点就是根节点，一棵树只有一个根节点。树结构的相关概念如下。

- 树的度：一个节点含有子树的个数称为该节点的度，一棵树中最大节点的度称为整棵树的度。
- 叶节点：度为 0 的节点称为叶节点。
- 根节点：没有父节点的节点就是根节点。
- 树的高度：从根节点开始，每多一级子节点，树的层次就+1，一棵树的最大层次数就是树的高度。
- 兄弟节点：具有相同父节点的子节点互称为兄弟节点。

树适合用来表达有层次关系的数据，比如一个公司的分支机构、计算机上的目录和文件结构，等等。如果树的子节点之间没有大小关系，则这样的树就称为无序树，也称自由树；如果树的子节点之间有大小关系，则这样的树就称为有序树。树通常也被认为是图的一种形式，是一种没有

环路的图，比如自由树可被视为一个连通的、无环路的无向图。根据每个节点的子节点的数量，又可以将树分为二叉树和多叉树，B 树就是一种典型的多叉树。

有序的二叉树也被称为二叉查找树（binary search tree）或二叉排序树（binary sort tree）。相对于普通的二叉树，二叉查找树有以下两个特点：

- 如果左子树不为空，则左子树上所有节点的值都小于根节点的值；
- 如果右子树不为空，则右子树上所有节点的值都大于根节点的值。

二叉查找树的特点是所有新插入的节点都是叶子节点，已经存在的节点的位置比较固定。二叉查找树查找操作的时间复杂度是 $O(\lg n)$，虽然线性有序表的查找操作时间复杂度也是 $O(\lg n)$（折半查找），但是线性有序表不能表达数据元素之间的关系。简单二叉查找树的插入操作都发生在叶节点，如果构造二叉查找树时依次插入的节点已经是有序的，则二叉树会退化为链表形状的单支树，如图 2-2 所示，在这种情况下，查找效率就会下降，查找操作的时间复杂度变成 $O(n)$。为了优化查找效率，就需要二叉查找树能够具有自平衡功能，保证二叉树始终是一棵平衡树。AVL 树和红黑树就是这样的自平衡二叉查找树，二者的区别在于维持树的自平衡的方法不一样。在算法设计时，只要有条件就应该优先使用 AVL 树和红黑树，避免简单二叉查找树可能存在的性能问题。

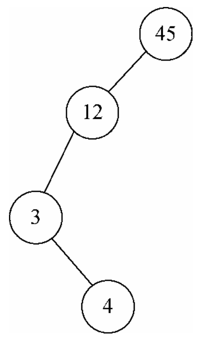


图 2-2 退化为链表的二叉排序树

二叉查找树在算法中的应用也很广泛，比如决策问题可以使用二叉查找树构造决策树，一些统计问题也可以使用二叉查找树的形式组织各种信息数据节点，其他的问题如果能将最终的结果转化成取舍问题，也可以使用二叉查找树。

多叉树的典型例子就是 B 树和各种 B 树的变形树，B 树是一种自平衡多叉查找树，对于一棵 M 阶 B 树来说，其每个非终端节点至少有 $\lceil M/2 \rceil$ 个子树，但是最多有 M 个子树，根节点如果不是终端节点，则至少有 2 个子树。每个节点有 N 个关键字，所有的终端节点都在统一的层次上，但是不带任何关键字信息。B+树是 B 树的一种变形，它与 B 树的差别在于对终端节点的处理，B+树的所有终端节点包含了全部关键字的信息，同时还有指向这些关键字所在节点的指针，并且终端节点按照关键字的大小排序，形成一个有序链接，因此对 B+树进行查找，可以从最小关键字开始顺序查找（遍历所有的终端节点），也可以从根节点开始遍历。B 树常用于文件管理系统和数据库系统，在算法设计时，如果遇到多路分支且有序的层次结构时，就可以考虑使用 B 树。

区间树是一种以区间为数据元素的红黑树，区间树的每个节点都表示一个区间，其关键字是区间的左端点，区间树支持所有的二叉查找树的基本操作，而且区间元素的插入和查找操作都可以在 $O(\lg n)$的时间内完成。区间树的查找不是精确查找，但是可以确定区间树上是否存在能完整覆盖给定的被查找区间的节点（区间）。区间树还支持获取某个子树的最大右端点的值的操作，

这个操作的意义在于当查找一个区间时，如果左子树的最大右端点的值小于当前查找区间的左端点的值，就说明左子树与当前查找的区间不存在重叠，需要转到右子树继续查找。区间树常用于区间查询相关的问题，比如判断区间之间是否存在重叠区域等问题。

线段树也是一种以区间为数据元素的二叉查找树，和区间树不同的是，线段树上的每个非叶子节点表示的区间范围是其子节点的区间范围之和。如果线段树中一个非叶子节点表示的区间是[a,b]，则它的左子树表示的区间就是[a,(a+b)/2]，右子树表示的区间就是[(a+b)/2,b]，由此可知，线段树是一棵平衡二叉树，一棵表示长度范围为[1,L]的线段树的高度是 $\lg(L)+1$。叶子节点表示不可再分的最小区间范围，如果将一个大区间分成 n 个小区间，则对应的线段树将有 n 个叶子节点，n 个叶子节点的小区间共同组成整个大区间。和区间树一样，线段树也可以用来做区间重叠性判断，除此之外，线段树在统计学相关的问题中应用也很广泛。比如一些统计信息，既需要查询在一个大的区间上的统计值，也需要查询在这个大范围中的某个小范围内的统计值，就可以应用线段树。举个例子，假如某杂志需要统计各个年龄段读者的比例，可以将统计的读者数量信息按照年龄区间组织成一棵线段树，如果一个节点的左边子节点是年龄在 20~30 岁之间的读者数量，右边子节点是年龄在 30~40 岁之间的读者数量，则这个节点表示的就是年龄在 20~40 岁之间的读者数量。

堆也是一种完全树，除了树的特征之外，堆的父节点和子节点还存在一些特殊关系。最大堆的每个节点的值都大于其子树上所有节点的值，最小堆的每个节点的值都小于其子树上所有节点的值。堆的插入和删除操作除了维持堆的完全树特征之外，还要维持节点之间的这个特殊关系，通常可以利用这一点实现一些特殊的功能，比如堆排序，就是利用堆的这个特殊性质，再比如经典的“求 n 个数中最大（或最小）的 m 个数的问题”，就是通过维护一个有 m 个节点的最大堆（或最小堆）来实现的。

解决一些与字符串相关的问题时，还会用到字典树，比如典型的前缀树和后缀树，图 2-3 就是一个字典树的例子。字典树以树的形式保存大量字符串（前缀或后缀），常被各种文本搜索算法用于文字和词频的统计。字典树的优点是利用字符串的公共前缀或后缀节约存储空间，查找过程中能减少无谓的完整字符串匹配，便于字符串的统计和查找。

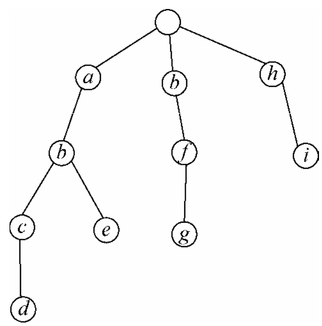


图 2-3　字典树示意图

2. 集合

简单来说，集合（set）是具有某种特性的事物的整体，构成集合的事物或对象称作集合的元素或成员。集合内的数据元素具有以下特征。

- 无序性：一个集合中每个元素的地位都是相同的，元素之间不存在有序关系，也没有类似树和图那样的复杂关系。
- 互异性：一个集合中每个元素只能出现一次，也就是说，集合内没有重复的元素。
- 确定性：集合的定义是确定的，根据这个定义可以明确判定一个对象是否属于这个集合，不存在模棱两可的情况。

集合的主要操作包括两部分，一部分是对集合元素的操作，包括插入和删除集合元素、判断一个元素是否属于集合等；另一部分是集合之间的关系运算，包括集合的求交集、并集运算以及求差运算等。集合作为一种数据元素的组织方式，在算法设计中应用也很广泛，比如第 19 章介绍数独游戏的算法时，就使用集合来管理每个小单元格的候选数列表。

3. 哈希表与映射

哈希表（hash）与映射（map）都是通过关键字（key）直接访问数据元素的值（value）的数据结构，二者的外部接口是一样的，但是不同的平台上对内部实现稍有差异。比如 C++的 STL 库中，`std::hash_map` 采用一个哈希函数实现从关键字到值的映射关系，数据的插入、删除和查找的时间复杂度都是 $O(1)$，而 `std::map` 则是采用红黑树实现关键字和值的存储，数据的插入、删除和查找的时间复杂度都是 $O(\lg(n))$。

哈希表的原理是通过一个哈希函数对关键字进行某种运算，得到对应的数据元素在表中的存储位置，然后访问其值，与普通的有序表查找相比，额外的哈希处理会造成数据访问的开销，但是哈希表的查找时间是固定的，不随哈希表中数据元素的增多而变化。普通有序表的查找时间复杂度是 $O(\lg(n))$，随着 n 的增大，查找时间也变长，当数据元素非常多的时候，哈希表的查找速度会比普通有序表快，这就是哈希表的优势。

现实生活中有很多采用“key-value”方式组织和存储数据的情况，学生成绩管理系统会通过一个唯一分配的学号建立与具体学生信息的映射关系，可以通过学号查询和管理学生信息，这个学号就是 key。

4. 图

图（graph）是一种特殊的数据组织方式，它不仅可以存储数据元素（对象），还可以存储数据元素之间的复杂关系。从直观上看，图由一些顶点和连接这些顶点的边组成，顶点描述数据元素，边描述数据元素之间的关系。图有很多种分类方式，根据边是否有方向，可将图分为有向图和无向图；根据任意两个顶点之间边的个数，可将图分为简单图和多重图；根据任意两个顶点之间的连通性，可将图分为连通图和非连通图。根据边的地位平等性，可分为带权图和不带权图。无论采用何种方法对图分类，定义图的方式基本上只有两种：二元组定义和三元组定义。对于图 G，如果 $V(G)$表示顶点集，$E(G)$表示边集，则(V, E)即为图的二元组定义。如果存在关联函数 I 将 $E(G)$中的每一条边映射到 $V(G)$中的两个顶点，即 $I(e) = (u, v)$，其中 u,v 属于 $V(G)$，且是 e 的两个顶点，则将(V, E, I)称为图的三元组定义。

图的存储常采用邻接矩阵（二维数组方式）和邻接表（链表或可变长数组）方式，对于有向图，有时候也采用十字链表方式。图的遍历是一个非常重要的操作，一般可采用深度优先搜索和广度优先搜索两种策略。深度优先搜索策略可以理解为树的中序（先根）遍历的推广，深度优先搜索的过程是：从图 G 的某个顶点 v 出发，先访问 v，然后选择一个与 v 相邻且没被访问过的顶点 v_i 访问，再从 v_i 出发选择一个与 v_i 相邻且未被访问的顶点 v_j 进行访问，依次遍历。如果当前被访问过的顶点的所有邻接顶点都已被访问，则退回到已被访问的顶点序列中最后一个拥有未被访

问的相邻顶点的顶点 v_k，从 v_k 出发按同样的方法向前遍历，直到图中的所有顶点都被访问过。从某个顶点开始深度优先搜索的伪代码实现如下：

```
//从第 v 个顶点出发递归地深度优先遍历图 G
void DFS(Graph G, int v)
{
    VisitFunction(v); //访问 v 点，并将其标记为访问过的节点;
    for_each(vi: v 的所有邻接点)//遍历 v 的所有邻接点
    {
        if(vi 没有被访问过)
            DFS(G, vi);
    }
}
```

广度优先搜索的过程是：首先访问初始点 v，接着依次访问 v 的所有未被访问过的邻接点 v_1，v_2，…，v_i，然后再按照 v_1，v_2，…，v_i 的次序，依次访问这些节点的邻接点（未被访问过的邻接点），依次遍历，直到图中所有和初始点 v_i 相邻的顶点都被访问过为止。为了保证 v 的所有邻接点被按照顺序依次处理，就需要使用队列来管理这些邻接点，这些都体现在广度优先搜索算法的伪代码中。从某个顶点开始广度优先搜索的伪代码如下：

```
void BFS(Graph G, int v)
{
    for_each(vi: v 的所有邻接点)//遍历 v 的所有邻接点
    {
        if(vi 没有被访问过)
        {
            VisitFunction(vi); //访问 v 点;
            EnQueue(Q, vi); //vi 入队列
        }
    }
    while(!QueueEmpty(Q))
    {
        DeQueue(Q, u);  //队头元素出队并置为 u
        BFS(G, u);
    }
}
```

现实生活中很多地方都用到了使用图的算法，比如你在地图软件中选择两个点，软件会给出连接这两个点之间的最佳路线，这就要用到图的连通性判断和最短路径搜索算法。当有多条路径可以连接两个点的时候，软件还可以根据不同道路的实时交通拥堵情况，选择最快捷的道路，其实也就是为每条道路设置不同的权重，然后进行带权图的最优路径搜索。交通规划常常需要用最小的成本（修最少的路）将不同的城市连接起来，保证每个城市之间都可以到达，就需要最小生成树算法。网络设备之间为了避免出现环路，也需要运行一个最小生成树协议（STP），也是图的应用。项目管理中计算项目活动的安排和求解项目关键路径功能，也会用到有向图的拓扑排序和关键路径算法，本书第 9 章就介绍了一个拓扑排序和关键路径算法在项目管理软件中得到应用的例子。

2.3 数据结构和数学模型与算法的关系

西方有句谚语：手里拿三年锤子，看什么都是钉子。如果一个人的本事就只会抡大锤，那他解决问题的方法就是拿锤子砸。有时候，工具可以决定思维。为什么要强调数据结构的重要性？因为数据结构是建立解决问题的数学模型的基础，如果不了解数据结构，就没有办法建立模型，或者建立的模型不合适，导致算法演化困难，甚至无法实现，第 1 章环形队列的使用就是最好的例子。掌握的数据结构越多，就相当于手中的工具越多，解决问题的思路就越宽。虽然我也建议大家尽量多地掌握复杂数据结构的实现原理，但是大多数情况下，这并不是必需的内容，只要知道数据结构的特性和对外的接口，就可以在算法设计时套用这些模式解决问题。了解这些数据结构的构造和操作原理固然重要，但是对于算法设计来说，灵活运用这些数据结构也很重要。

另一方面，我们也强调数学模型，但是这并是不单纯地说所有问题都是数学问题，而是因为计算机善于处理数学问题。或者我们可以将其描述为更笼统的概念，比如计算机模型。但是实质都是一样的，如果要计算机解决问题，就必须用计算机能理解的方式描述问题。建立问题的数学模型实际上是对问题的一种抽象的表达，通常也需要伴随着一些合理的假设，其目的就是对问题进行简化，抓住主要因素，舍弃次要因素，逐步用更精确的语言描述问题，最终过渡到用计算机语言能够描述问题为止。让我们来看两个通过建立抽象的模型，将看似复杂的问题简化并最终解决的例子吧。

一个工程项目经过层层结构分解最终得到一系列具体的活动，这些活动之间往往存在复杂的依赖关系，如何安排这些活动的开始顺序，使得项目能够顺利完成是个艰巨的任务。但是如果能把这个问题转化成有向图，图的顶点就是活动，顶点之间的有向边代表活动之间的前后关系，则只需要使用简单的有向图拓扑排序算法就可以解决这个问题。一个工程分解出这么多的活动，每个活动的时间都不一样，如何确定工程的最短完工时间？工程的最短完工时间取决于这些活动中时间最长的那条关键活动路径，从成百上千个活动中找出关键路径看似是个无法入手的问题，但是如果将问题转化为有向图，顶点代表事件，边代表活动，边的权代表活动时间，则可以利用有向图的关键路径算法解决问题。

用三个容积分别为 3 升、8 升和 5 升的水桶如何获得 4 升水的问题，是一个经典的智力游戏。如果让计算机像人一样思考并解决这个问题有点困难，但是如果转换思维，将三个水桶中当前的水量定义为一个状态，将倒水定义为一个驱动状态转换的动作，则这个问题就转换为水桶状态的穷举搜索问题。在解空间中用穷举的方法遍历所有可能的解，并找到最终合法的解是解决最优解问题的常用数学模型，只要想到了这个数学模型，这个问题就迎刃而解了，在本书的第 5 章，你将会看到如何利用这个数学模型解决倒水问题的算法实现。

你可以设计数学模型，但是有时候你也可以像使用模式一样使用那些经典的或常用的数学模型，或者根据不同对象的某些相似性，借用已知领域的数学模型。当我们解决未知的问题时，常常把已知的旧问题当作基础或经验来源。正如艾萨克·牛顿说的那样：“如果我看得比别人远，那是因为我站在巨人的肩膀上。”从根本上讲，把未知的问题转化成已知问题，然后再用已知的方法解决已知问题，是解决未知问题的基础手段。但是，如何将一个未知的问题转化为我们熟知

的数学模型是一个复杂而艰难的过程，完成这个过程需要相当多的经验积累，同时也是算法设计中最有趣味的部分，下面来看一个算法几何的例子。

判断 n 个矩形之间是否存在包含关系是经典的算法几何问题，按照一般的思路，应该是 n 个矩形两两进行包含判断。但是很显然，这个简单的方法需要 $n(n-1)$次矩形包含判断，时间复杂度是 $O(n^2)$。如果知道区间树的概念，就可以将这个问题转化为区间树的查询问题。首先根据矩形的几何位置，利用水平边和垂直边分别构造两颗区间树（根据矩形的几何特征，只需要处理一条水平边和一条垂直边即可），然后将 n 个矩形的水平边作为被查找元素，依次在水平边区间树中查找，如果找到其他矩形的水平边完整覆盖被查找矩形的水平边，则在垂直边区间树上进一步判断该矩形的垂直边被覆盖的情况。如果存在被查找的矩形的水平边和垂直边都被同一个矩形的水平边和垂直边覆盖的情况，则说明这两个矩形存在包含关系。采用区间树的算法复杂度是 $O(n\lg(n))$，额外的开销是建立区间树的开销，但是只要 n 足够大，这个算法仍然比简单比较法高效。

数据结构是算法的基本工具，采用什么数据结构由算法的数学模型决定，但是各不相同的数据结构自身的一些特点反过来也会影响数学模型的选择。数学模型是对问题域的高度抽象，而数据结构又是承载数学模型的基础。在简单的算法中，数学模型的定义有时候可能简化成数据结构的定义，即便是复杂的数学模型最终也是要使用相应定义的数据结构来承载，所以这二者是亲密无间的孪生兄弟，它们共同决定了算法的成败。

2.4 总结

本章介绍了三部分内容，分别是程序的基本结构、常用数据结构及应用方式以及数据结构和数学模型与算法的关系。程序的基本结构不需要特别强调，因为这应该是每一个程序员的本能。对常用的数据结构的介绍也着眼于这些数据结构的特点和适用的场合，重点不是原理，而是如何在不同场合下灵活使用这些数据结构。数据结构和数学模型与算法的关系是不言而喻的，二者是密不可分的，或者是同一事物的两个方面，它们都是算法的基础。像 2.3 节中的例子，还有很多，这就是将问题转化为适当的数学模型后体现出来的威力。对这些常用的数据结构的了解和掌握，在很大程度上决定了建立数学模型的能力。

2.5 参考资料

[1] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[2] 维基百科：http://zh.wikipedia.org/wiki/树_(数据结构)

[3] 维基百科：http://zh.wikipedia.org/wiki/图_(数据结构)

[4] Levitin A. 算法设计与分析基础. 潘彦译. 北京：清华大学出版社，2007

[5] Kleigberg J, Tardos E. *Algorithm Design*. Addison-Wesley, 2005

第 3 章 算法设计的常用思想

本书第 1 章将算法设计描述为像小宇宙爆发一样的智力活动的结果，其实是不妥当的。《算法设计》一书的作者将算法设计定义为一个这样的设计过程：从广泛的计算机应用中提出问题开始，建立在对算法设计技术理解的基础上，并最终发展成对这些问题的有效解决[5]。从这个意义上理解，算法确实是一次智力活动的结果，但是并不是毫无章法的爆发，它应该是遵循一定规律的智力活动。首先，它需要一些基础性的知识作为这种智力活动的着力点，比如数据结构。其次，它需要对问题域做充分的分析和研究，高度概括并抽象出问题的精确描述，也就是各种建立数学模型的方法。最后，有一些常用的模式和原则，可以作为构造算法的选择项，有人称之为算法设计方法，我建议称之为算法设计模式或算法设计思想，以便将其与一些具体的算法名称区分开。

模式作为算法演进的一些固定的思路，提供了一些构造算法的常用思想，本章将介绍几种典型的算法设计模式，每种模式都适合解决一类特定的问题，每种模式都配合一个具体的例子做说明，通过实例介绍这种设计思想的适用原则。

3.1 贪婪法

贪婪法（greedy algorithm），又称贪心算法，是寻找最优解问题的常用方法。这种方法模式一般将求解过程分成若干个步骤，在每个步骤都应用贪心原则，选取当前状态下最好的或最优的选择（局部最有利的选择），并以此希望最后堆叠出的结果也是最好或最优的解。贪婪法的每次决策都以当前情况为基础并根据某个最优原则进行选择，不从整体上考虑其他各种可能的情况。一般来说，这种贪心原则在各种算法模式中都会体现，单独作为一种方法来说明，是因为贪婪法对于特定的问题是非常有效的方法。

贪婪法和动态规划法以及分治法一样，都需要对问题进行分解，定义最优解的子结构。但是，贪婪法与其他方法最大的不同在于，贪婪法每一步选择完之后，局部最优解就确定了，不再进行回溯处理，也就是说，每一个步骤的局部最优解确定以后，就不再修改，直到算法结束。因为不进行回溯处理，贪婪法只在很少的情况下可以得到真正的最优解，比如最短路径问题、图的最小生成树问题。大多数情况下，由于选择策略的“短视”，贪婪法会错过真正的最优解，得不到问

题的真正答案。但是贪婪法简单高效，省去了为找最优解可能需要的穷举操作，可以得到与最优解比较接近的近似最优解，通常作为其他算法的辅助算法使用。

3.1.1 贪婪法的基本思想

贪婪法的基本设计思想有以下三个步骤。

(1) 建立对问题精确描述的数学模型，包括定义最优解的模型；

(2) 将问题分解为一系列子问题，同时定义子问题的最优解结构；

(3) 应用贪心原则确定每个子问题的局部最优解，并根据最优解的模型，用子问题的局部最优解堆叠出全局最优解。

定义最优解的模型通常和定义子问题的最优解结构是同时进行的，最优解的模型一般都体现了最优解子问题的分解结构和堆叠方式。对于子问题的分解有多种方式，有的问题可以按照问题的求解过程一步一步地进行分解，每一步都在前一步的基础上选择当前最好的解，每做一次选择就将问题简化为一个规模更小的子问题，当最后一步的求解完成后就得到了全局最优解。还有的问题可以将问题分解成相对独立的几个子问题，对每个子问题求解完成后再按照一定的规则（比如某种公式或计算法则）将其组合起来得到全局最优解。

这里说的定义子问题分解和子问题的最优解结构可能有点抽象，我们来看一个具体的例子。找零钱是一个经典的例子，假如某国发行的货币有 25 分、10 分、5 分和 1 分四种硬币，假如你是售货员，你要找给客户 41 分钱的硬币，如何安排能使得找给客人的钱正确，但是硬币个数最少。这个问题的子问题定义就是从四种币值的硬币中选择一枚，使这个硬币的币值和其他已经选择的硬币的币值总和不超过 41 分钱。子问题的最优解结构就是在之前的步骤中已经选择的硬币加上当前选择的一枚硬币，当然，选择的策略是贪婪策略，即在币值总和不超过 41 的前提下选择币值最大的那种硬币。按照这个策略，第一次选择 25 分的硬币一枚，第二次选择 10 分的硬币一枚，第三次选择 5 分的硬币一枚，第四次选择 1 分的硬币一枚，总共需要 4 枚硬币。

上面的例子得到的确实是一个最优解，但是很多情况下贪婪法都不能得到最优解。同样以找零钱为例，假如某国发行的货币是 25 分、20 分、5 分和 1 分四种硬币，这时候找 41 分钱的最优策略是 2 枚 20 分的硬币加一枚 1 分硬币共 3 枚硬币，但是用贪婪法得到的结果却是 1 枚 25 分硬币，三枚 5 分硬币和一枚 1 分硬币共 5 枚硬币。

3.1.2 贪婪法的例子：0-1 背包问题

本节介绍一个贪婪法的经典例子——0-1 背包问题：有 N 件物品和一个承重为 C 的背包（也可定义为体积），每件物品的重量是 w_i，价值是 p_i，求解将哪几件物品装入背包可使这些物品在重量总和不超过 C 的情况下价值总和最大。背包问题（knapsack problem）是此类组合优化的 NP 完全问题的统称，比如货箱装载问题、货船载物问题等，因问题最初来源于如何选择最合适的物品装在背包中而得名。这个问题隐含了一个条件，每个物品只有一件，也就是限定每件物品只能

选择 0 个或 1 个，因此又被称为 0-1 背包问题。

来看一个具体的例子，有一个背包，最多能承载重量为 C=150 的物品，现在有 7 个物品（物品不能分割成任意大小），编号为 1 ~ 7，重量分别是 w_i=[35,30,60,50,40,10,25]，价值分别是 p_i=[10,40,30,50,35,40,30]，现在从这 7 个物品中选择一个或多个装入背包，要求在物品总重量不超过 C 的前提下，所装入的物品总价值最高。这个问题的子问题可以按照选择物品装入背包的过程按部就班地一步一步分解，将子问题定义为在被背包容量还有 C' 的情况下，选择一个物品装入背包。C' 的初始值就是 150，假如选择了一个重为 35 的物品，则子问题就变成在背包容量 C' 是 115 的情况下，从剩下的 6 件物品中选择一个物品，这样每选择一个物品就相当于子问题的规模减小了。

那么如何选择物品呢？这就是贪婪策略的选择问题。对于本题，常见的贪婪策略有三种。第一种策略是根据物品价值选择，每次都选价值最高的物品。根据这个策略最终选择装入背包的物品编号依次是 4、2、6、5，此时包中物品总重量是 130，总价值是 165。第二种策略是根据物品重量选择，每次都选择重量最轻的物品。根据这个策略最终选择装入背包的物品编号依次是 6、7、2、1、5，此时包中物品总重量是 140，总价值是 155。第三种策略是定义一个价值密度的概念，每次选择都选价值密度最高的物品。物品的价值密度 s_i 定义为 p_i/w_i，这 7 件物品的价值密度分别为 s_i=[0.286,1.333,0.5,1.0,0.875,4.0,1.2]。根据这个策略最终选择装入背包的物品编号依次是 6、2、7、4、1，此时包中物品的总重量是 150，总价值是 170。

根据前文的分析结果，我们给出贪婪法解决背包问题的算法实现。首先定义背包问题的数据结构，根据问题描述，可以直接知道每个物品有两个属性，分别是重量和价值。此外，每个物品只能被选择一次，因此还需要给每个物品增加一个选择状态的属性，因此物品的数据结构定义如下：

```
typedef struct tagObject
{
    int weight;
    int price;
    int status; //0:未选中；1:已选中；2:已经不可选
}OBJECT;
```

需要特别说明的是状态值为 2 的情况，这种情况表示用当前策略选择的物品导致总重量超过背包承重量，在这种情况下，如果放弃这个物品，按照策略从剩下的物品中再选一个，有可能就能满足背包承重的要求。因此，设置了一个状态 2，表示当前选择物品不合适，下次选择也不要再选这个物品了。接下来是背包问题的定义，背包问题包括两个属性，一个是可选物品列表，一个是背包总的承重量。简单定义背包问题数据结构如下：

```
typedef struct tagKnapsackProblem
{
    std::vector<OBJECT> objs;
    int totalC;
}KNAPSACK_PROBLEM;
```

GreedyAlgo()函数是贪婪算法的主体结构，包括子问题的分解和选择策略的选择都在这个函数中。正如函数所展示的那样，它可以作为此类问题的一个通用解决思路。

```
void GreedyAlgo(KNAPSACK_PROBLEM *problem, SELECT_POLICY spFunc)
{
    int idx;
    int ntc = 0;

    //spFunc 每次选最符合策略的那个物品，选后再检查
    while((idx = spFunc(problem->objs, problem->totalC - ntc)) != -1)
    {
        //所选物品是否满足背包承重要求?
        if((ntc + problem->objs[idx].weight) <= problem->totalC)
        {
            problem->objs[idx].status = 1;
            ntc += problem->objs[idx].weight;
        }
        else
        {
            //不能选这个物品了，做个标记后重新选
            problem->objs[idx].status = 2;
        }
    }

    PrintResult(problem->objs);
}
```

spFunc 参数是选择策略函数的接口，通过替换这个参数，可以实现上文提到的三种贪婪策略，分别得到各种贪婪策略下得到的解。以第一种策略为例，可以这样实现：

```
int Choosefunc1(std::vector<OBJECT>& objs, int c)
{
    int index = -1;
    int mp = 0;
    for(int i = 0; i < static_cast<int>(objs.size()); i++)
    {
        if((objs[i].status == 0) && (objs[i].price > mp))
        {
            mp = objs[i].price;
            index = i;
        }
    }

    return index;
}
```

看起来第三种策略取得了最好的结果，和动态规划方法得到的最优结果是一致的。但是实际上，这只是对这组数据的验证结果而已，如果换一组数据，结果可能完全相反。当然，对于一些能够证明贪婪策略得到的就是最优解的问题，应用贪婪法可以高效地求得结果，比如求最小生成树的 Prim 算法和 Kruskal 算法。大多数情况下，贪婪法只能得到比较接近最优解的近似的最优解，但是作为一种启发式辅助方法，它常用于其他算法中，比如 Dijkstra 的单源最短路径算法。事实

上，在任何算法中，只要在某个阶段使用了只考虑局部最优情况的选择策略，都可以理解为使用了贪婪算法。

3.2　分治法

分治，顾名思义，分而治之。分治法（divide and conquer）也是一种解决问题的常用模式，分治法的设计思想是将无法着手解决的大问题分解成一系列规模较小的相同问题，然后逐个解决小问题，即所谓的分而治之。分治法产生的子问题与原始问题相同，只是规模减小，反复使用分治方法，可以使得子问题的规模不断减小，直到能够被直接求解为止。

分治法作为算法设计中一个古老的策略，在很多问题中得到了广泛的应用，比如最大、最小问题（例如在一堆形状相同的物品中找出最重或最轻的那一个），矩阵乘法、大整数乘法以及排序（例如快速排序和归并排序）。除此之外，这个技巧也是许多高效算法的基础，比如快速傅里叶变换算法和 Karatsuba 乘法算法。

应用分治法，一般出于两个目的：一是通过分解问题，使无法着手解决的大问题变成容易解决的小问题；二是通过减小问题的规模，降低解决问题的复杂度（或计算量）。给 1000 个数排序，可能会因为问题的规模太大而无从下手，但是如果减小这个问题的规模，将问题一分为二，变成分别对两个拥有 500 个数的序列排序，然后再将两个排序后的序列合并成一个就得到了 1000 个数的排序结果。对 500 个数排序仍然无法下手，需要继续分解，直到最后问题的规模变成 2 个数排序的时候，只需要一次比较就可以确定顺序。这正是快速排序的实现思想，通过减小问题的规模使问题由难以解决变得容易解决。计算 N 个采样点的离散傅里叶变换，需要做 N^2 次复数乘法，但是将其分解成两个 $N/2$ 个采样点的离散傅里叶变换，则只需要做 $(N/2)^2+(N/2)^2 = N^2/2$ 次复数乘法，做一次分解就使得计算量减少了一半，这正是快速傅里叶变换的实现思想，通过减小问题的规模减少计算量，降低问题的复杂度。

3.2.1　分治法的基本思想

很多情况下，分治法都会使用递归的方式对问题逐级分解，但是在每个子问题的层面上，分治法基本上可以归纳为以下三个步骤。

(1) 分解：将问题分解为若干个规模较小，相互独立且与原问题形式相同的子问题，确保各个子问题的解具有相同的子结构。

(2) 解决：如果上一步分解得到的子问题可以解决，则解决这些子问题，否则，对每个子问题使用和上一步相同的方法再次分解，然后求解分解后的子问题，这个过程可能是一个递归的过程。

(3) 合并：将上一步解决的各个子问题的解通过某种规则合并起来，得到原问题的解。

分治法的实现模式可以是递归方式，也可以是非递归方式，一般采用递归方式的算法模式可以用伪代码描述为：

```
T DivideAndConquer(P)
{
    if(P 可以直接解决)
    {
        T <- P 的结果;
        return T;
    }

    将 P 分解为子问题{P1, P2,..., Pn};
    for_each(Pi : {P1, P2,..., Pn})
    {
        ti <- DivideAndConquer(Pi); //递归解决子问题 Pi
    }
    T <- Merge(t1, t2,...,tn); //合并子问题的解

    return T;
}
```

分治法的难点是如何将子问题分解，并且将子问题的解合并出原始问题的解。针对不同的问题，通常有不同的分解与合并的方式。

- 快速排序算法的分解思想是选择一个标兵数，将待排序的序列分成两个子序列，其中一个子序列中的数都小于标兵数，另一个子序列中的数都大于标兵数，然后分别对这两个子序列排序，其合并思想就是将两个已经排序的子序列一前一后拼接在标兵数前后，组成一个完整的有序序列。
- 快速傅里叶变换的分解思想是将一个 N 点离散傅里叶变换，按照奇偶关系分成两个 $N/2$ 点离散傅里叶变换,其合并思想就是将两个 $N/2$ 点离散傅里叶变换的结果按照蝶形运算的位置关系重新排列成一个 N 点序列。
- Karatsuba 乘法算法的分解思想是将 n 位大数分成两部分：$a+b$，其中 a 是整数幂，然后利用乘法的分解公式：$(a+b)(c+d)=ac+ad+bc+bd$，将其分解为四次小规模大数的乘法计算，然后利用一个小技巧将其化解成三次乘法和少量移位操作。最终结果的合并思想就是用几次加法对小规模乘法的结果进行求和，得到原始问题的解。

由以上例子可知，分治法最难也是最灵活的部分就是对问题的分解和结果的合并。对于一个未知的问题，只要能找到对子问题的分解方式和结果的合并方式，应用分治法就可以迎刃而解。而在数学上，只要是能用数学归纳法证明的问题，一般都可以应用分治法解决，这也是一个应用分治法的强烈信号。

3.2.2 递归和分治，一对好朋友

递归作为一种算法的实现方式，与分治法天生是一对好朋友。问题的分解肯定不是一步到位的，需要反复使用分治手段，在多个层次上层层分解，这种分解的方法很自然地导致了递归方式的使用。从算法实现的角度看，分治法得到的子问题和原问题是相同的，当然可以用相同的函数来解决，区别只在于问题的规模和范围不同。通过特定的函数参数安排，使得同一个函数可以解

决不同规模的相同问题，这就是递归方法的基础。

以快速排序为例，如果把待排序的序列作为问题的话，那么子问题的规模就可以定义为子序列在原始序列中的起始位置。对此一般化之后，原始问题和子问题的描述就统一了，都是原始序列+起始位置，原始问题的起始位置就是[1,*n*]，子问题的起始位置就是[1,*n*]中的某一个子区间，由此一来，递归的接口就明确了：

```
void quick_sort(int *arElem, int p, int r)
```

其中，*p* 和 *r* 分别是子序列在 arElem 中的起始位置，有了子问题的递归定义接口，快速排序的算法实现也就水到渠成了：

```
void quick_sort(int *arElem, int p, int r)
{
    if(p < r)
    {
        int mid = partion(arElem, p, r);
        quick_sort(arElem, p, mid - 1);
        quick_sort(arElem, mid + 1, r);
    }
}
```

不用递归是不是就不能用分治法了？当然不是，快速傅里叶变换算法就没有用递归。很多算法都有自己的非递归实现方式，是否使用了递归方法不是判断是否是分治法的必要条件。即便是一些使用了递归方法的算法，也都可以用一个自己构造的栈将其改编为非递归方法，比如快速排序就有很多用栈实现的非递归方法。Robert Sedgewick 在其著作 *Algorithm in C* 一书中就给出了一种快速排序的非递归高效算法。有兴趣的读者可阅读此书，了解一下算法的实现。

3.2.3 分治法的例子：大整数 Karatsuba 乘法算法

两个 n 位大整数相乘，普通乘法算法的时间复杂度一般是 $O(n^2)$。但是 Anatolii Alexeevitch Karatsuba 博士在 1960 年提出了一种时间复杂度是 $O(3n^{1.585})$（$1.585=\log_2 3$）的快速乘法算法，这就是 Karatsuba 乘法算法。该算法就是利用了分治法的思想，将 n 位大整数分解成两个接近 $n/2$ 位的大整数，通过 3 次 $n/2$ 位大整数的乘法和少量加法操作，避免了直接进行 n 位大整数乘法计算，有效地降低了乘法计算的计算量。

Karatsuba 乘法算法的原理非常简单，假如有两个 n 位的 M 进制大整数 x，y，利用一个小于 n 的正数 k（通常 k 的取值为 $n/2$ 左右），将 x 和 y 分解为两个部分：

$$x = x_1M^k + x_0$$

$$y = y_1M^k + y_0$$

则 x 和 y 的乘积可计算为：

$$xy = (x_1M^k + x_0)(y_1M^k + y_0) = x_1y_1M^{2k} + (x_1y_0 + x_0y_1)M^k + x_0y_0$$

这样就将 x 和 y 的乘法计算转化成四次较小规模的乘法计算和少量的加法计算，其中 M^{2k} 和 M^k 的计算都可以通过移位高效地处理。不过上述操作还可以继续优化，我们令 $z_0=x_0y_0$，$z_1=x_1y_0+x_0y_1$，$z_2=x_1y_1$，则 xy 的乘积可表示为：

$$xy = z_2M^{2k} + z_1M^k + z_0$$

计算 z_1 需要两次乘法，对 z_1 的计算可以优化为：

$$z_1 = (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$$

由于 z_0 和 z_2 都已经计算过了，因此就只需一次乘法，辅助两次加法和两次减法即可计算出 z_1。

根据以上分析，Karatsuba 乘法算法对子问题的分解就是将大整数分成高位和低位两个部分，然后利用优化后的计算公式计算出最后的结果，而这个计算公式就是结果的合并部分。本书的第 17 章会介绍大整数相关算法，其中的 CBigInt 类就是一个 n 位的 2^{32} 进制大数，可以理解为 $M=2^{32}$，利用 CBigInt 类的实现，我们给出 n 位的 2^{32} 进制大数的 Karatsuba 乘法算法实现：

```
CBigInt Karatsuba(const CBigInt& mul1, const CBigInt& mul2)
{
    //1 位大整数，直接计算，这也是递归的终止条件
    if((mul1.GetBigNCount() == 1) || (mul2.GetBigNCount() == 1))
    {
        return mul1 * mul2;
    }
    //问题分解
    CBigInt high1,high2,low1,low2;
    unsigned int k = max(mul1.GetBigNCount(), mul2.GetBigNCount()) / 2;
    high1 = mul1;
    high1.GetRightBigN(k, low1);
    high1.ShiftRightBigN(k);
    high2 = mul2;
    high2.GetRightBigN(k, low2);
    high2.ShiftRightBigN(k);
    CBigInt z0 = Karatsuba(low1, low2);
    CBigInt z1 = Karatsuba((low1 + high1), (low2 + high2));
    CBigInt z2 = Karatsuba(high1, high2);
    //结果合并
    CBigInt zk = z1 - z2 - z0;
    z2.ShiftLeftBigN(2 * k);
    zk.ShiftLeftBigN(k);

    return (z2 + zk + z0);
}
```

CBigInt::GetBigNCount()函数计算大整数的位数，是以 2^{32} 为进制的位数，k 作为分解位数，是两个大数的位数的一半，将高位和低位分别取出进行计算。其中三次乘法计算继续利用分治的思想进行处理，当问题的规模减少到 1 位大整数时就可以直接计算了，递归调用 Karatsuba()函数体现了这种思想。

3.3　动态规划

动态规划（dynamic programming）是解决多阶段决策问题常用的最优化理论，该理论由美国数学家 Bellman 等人在 1957 年提出，用于研究多阶段决策过程的优化问题。该理论提出后，立即在数学、计算机科学、经济管理和工程技术领域得到了广泛的应用，例如最短路线、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法往往比朴素的方法更高效。动态规划方法的原理就是把多阶段决策过程转化为一系列的单阶段决策问题，利用各个阶段之间的递推关系，逐个确定每个阶段的最优化决策，最终堆叠出多阶段决策的最优化决策结果。

动态规划比穷举高效，这一点在很多情况下都得到了印证，这常常给人一种错觉，以为它是高效的多项式时间算法，但事实并非如此。应用动态规划法解题的效率，取决于问题的类型，并不是任何情况下使用动态规划法都有最高的效率。比如很多 NP 问题，也可以设计用动态规划法解决。在这种情况下，动态规划法就不是一种多项式时间的方法，而是一种穷举，其效率就是指数时间复杂度。反过来理解这个问题也是一样的，很多 NP 问题可以使用动态规划方法求解，如果简单地认为动态规划是一种多项式方法，那难道这些问题就不再是 NP 问题了？

Kleigberg 在他的《算法设计》一书中也对这个问题进行了讨论，他认为动态规划法通过将问题细分为一系列子问题，从而隐含地探查了所有可行解的空间，于是我们可以从某种程度上把动态规划看作接近暴力搜索边缘的危险操作。对于多项式时间的问题，动态规划法可能得到多项式时间复杂度的高效算法，但是对于 NP 问题，动态规划法也只能得到指数时间复杂度的算法。Kleigberg 认为，动态规划对子问题的处理方式使得它可以遍历问题可行解的指数规模的集合，甚至可以在没有明确地检查所有解的情况下就做到这一点。可以认为这是因为动态规划拥有比穷举更高效的剪枝判断，这是一种对重叠子问题（子问题包含子子问题）处理的内在机制。

每种方法都有自身的局限性，动态规划法也不是万能的。动态规划适合求解多阶段（状态转换）决策问题的最优解，也可用于含有线性或非线性递推关系的最优解问题，但是这些问题都必须满足最优化原理和子问题的“无后向性”。

- 最优化原理：最优化原理其实就是问题的最优子结构的性质，如果一个问题的最优子结构是不论过去状态和决策如何，对前面的决策所形成的状态而言，其后的决策必须构成最优策略。也就是说，不管之前决策是否是最优决策，都必须保证从现在开始的决策是在之前决策基础上的最优决策，则这样的最优子结构就符合最优化原理。
- 无后向性（无后效性）：所谓“无后向性”，就是当各个阶段的子问题确定以后，对于某个特定阶段的子问题来说，它之前的各个阶段的子问题的决策只影响该阶段的决策，对该阶段之后的决策不产生影响，也就是说，每个阶段的决策仅受之前决策的影响，但是不影响之后各阶段的决策。

3.3.1　动态规划的基本思想

和分治法一样，动态规划解决复杂问题的思路也是对问题进行分解，通过求解小规模的子问

题再反推出原问题的结果。但是动态规划分解子问题不是简单地按照“大事化小”的方式进行的，而是沿着决策的阶段划分子问题，决策的阶段可以随时间划分，也可以随着问题的演化状态划分。分治法要求子问题是互相独立的，以便分别求解并最终合并出原始问题的解，但是动态规划法的子问题不是互相独立的，子问题之间通常有包含关系，甚至两个子问题可以包含相同的子子问题。比如，子问题 A 的解可能由子问题 C 的解递推得到，同时，子问题 B 的解也可能由子问题 C 的解递推得到。对于这种情况，动态规划法对子问题 C 只求解一次，然后将其结果保存在一张表中（此表也称为备忘录），避免每次遇到这种情况都重复计算子问题 C 的解。除此之外，动态规划法的子问题还要满足“无后向性”要求。

动态规划法不像贪婪法或分治法那样有固定的算法实现模式，作为解决多阶段决策最优化问题的一种思想，它没有具体的实现模式，可以用带备忘录的递归方法实现，也可以根据堆叠子问题之间的递推公式用递推的方法实现。但是从算法设计的角度分析，使用动态规划法一般需要四个步骤，分别是定义最优子问题、定义状态、定义决策和状态转换方程以及确定边界条件，这四个问题解决了，算法也就确定了。接下来就结合几个实例分别介绍这四个步骤，这几个例子分别是《算法导论》一书[1]中介绍的装配站问题、前文提到的背包问题以及经典的最长公共子序列问题（longest common subsequence）。

1. 定义最优子问题

定义最优子问题，也就是确定问题的优化目标以及如何决策最优解，并对决策过程划分阶段。所谓阶段，可以理解为一个问题从开始到解决需要经过的环节，这些环节前后关联。划分阶段没有固定的方法，根据问题的结构，可以按照时间顺序划分阶段，也可以按照问题的演化状态划分阶段。阶段划分以后，对问题的求解就变成对各个阶段分别进行最优化决策，问题的解就变成按照阶段顺序依次选择的一个决策序列。

装配站问题的阶段划分比较清晰，把工件从一个装配站移到下一个装配站就可以看作是一个阶段，其子问题就可以定义为从一个装配站转移到下一个装配站，直到最后一个装配站完成工件组装。对于背包问题，每选择装一个物品就可以看作一个阶段，其子问题就可以定义为每次向包中装一个物品，直到超过背包的最大容量为止。最长公共子序列问题可以按照问题的演化状态划分阶段，这需要首先定义状态，有了状态的定义，只要状态发生了变化，就可以认为是一个阶段。

2. 定义状态

状态既是决策的对象，也是决策的结果，对于每个阶段来说，对起始状态施加决策，使得状态发生改变，得到决策的结果状态。初始状态经过每一个阶段的决策（状态改变）之后，最终得到的状态就是问题的解。当然，不是所有的决策序列施加于初始状态后都可以得到最优解，只有一个决策序列能得到最优解。状态的定义是建立在子问题定义的基础上的，因此状态必须满足“无后向性”要求。必要时，可以增加状态的维度，引入更多的约束条件，使得状态定义满足“无后向性”要求。

装配站问题的实质就是在不同的装配线之间选择装配站，使得工件装配完成的时间最短，其

状态 $s[i,j]$就可以定义为通过第 i 条装配线的第 j 个装配站所需要的最短时间。背包问题本身是个线性过程，但是如果简单将状态定义为装入的物品编号，也就是定义 $s[i]$为装入第 i 件物品后获得的最大价值，则子问题无法满足“无后向性”要求，原因是之前的任何一个决策都会影响到所有的后序决策（因为装入物品后背包容量发生了变化），因此需要增加一个维度的约束。考虑到每装入一个物品，背包的剩余容量就会减少，故而选择将背包容量也包含的状态定义中。最终背包问题的状态 $s[i,j]$ 定义为将第 i 件物品装入容量为 j 的背包中所能获得的最大价值。对于最长公共子序列问题，如果定义 str1[1…i]为第一个字符串前 i 个字符组成的子串，定义 str2[1…j]为第二个字符串的前 j 个字符组成的子串，则最长公共子序列问题的状态 $s[i,j]$ 定义为 str1[1…i]与 str2[1…j]的最长公共子序列长度。

3. 定义决策和状态转换方程

定义决策和状态转换方程。决策就是能使状态发生转变的选择动作，如果选择动作有多个，则决策就是取其中能使得阶段结果最优的那一个。状态转换方程是描述状态转换关系的一系列等式，也就是从 n-1 阶段到 n 阶段演化的规律。状态转换取决于子问题的堆叠方式，如果状态定义得不合适，就会导致子问题之间没有重叠，也就不存在状态转换关系了。没有状态转换关系，动态规划也就没有意义了，实际算法就退化为像分治法那样的朴素递归搜索算法。

对于装配站问题，其决策就是选择在当前工作线上的下一个工作站继续装配，或者花费一定的开销将其转移到另一条工作线上的下一个工作站继续装配。如果定义 $a[i,j]$为第 i 条工作线的第 j 个装配站需要的装配时间，$k[i,j]$为从另一条工作线转移到第 i 条装配线的第 j 个装配站需要的转移开销，则装配站问题的状态转换方程可以描述为：

$$s[1,j] = \min(s[1,j-1]+a[1,j],\ s[2,j-1]+k[1,j]+a[1,j])$$

$$s[2,j] = \min(s[2,j-1]+a[2,j],\ s[1,j-1]+k[2,j]+a[2,j])$$

背包问题的决策很简单，就是判断装入第 i 件物品获得的收益最大还是不装入第 i 件物品获得的收益最大。如果不装入第 i 件物品，则背包内物品的价值仍然是 $s[i-1,j]$状态，如果装入第 i 件物品，则背包内物品的价值就变成 $s[i, j-Vi]+P_i$状态，其中 V_i和 P_i分别是第 i 件物品的容积和价值，决策的状态转换方程就是：

$$s[i,j] = \max(s[i-1,j],\ s[i,j-V_i]+P_i)$$

最长公共子序列问题的决策方式就是判断 str1[i]和 str2[j]的关系，如果 str1[i]与 str2[j]相同，则公共子序列的长度应该是 s[i−1，j−1]+1，否则就分别尝试匹配 str1[1⋯i−1]与 str2[1⋯j]的最长公共子串，以及 str1[1⋯i]与 str2[1⋯j−1]的最长公共子串，然后取二者中较大的那个值作为 $s[i,j]$的值。最长公共子序列问题的状态转换方程就是：

$$s[i,j] = s[i-1,j-1] + 1 \qquad ;\ \text{str1}[i]\text{与 str2}[j]\text{相同}$$

$$s[i,j] = \max(s[i,j-1],\ s[i-1,j])\ ;\ \text{str1}[i]\text{与 str2}[j]\text{不相同}$$

4. 确定边界条件

对于递归加备忘录方式（记忆搜索）实现的动态规划方法，边界条件实际上就是递归终结条件，无需额外的计算。对于使用递推关系直接实现的动态规划方法，需要确定状态转换方程的递推式的初始条件或边界条件，否则无法开始递推计算。

对于装配站问题，初始条件就是工件通过第一个装配站的时间，对于两条装配线来说，工件通过第一个装配站的时间虽然不相同，但是都是确定的值，就是移入装配线的开销加上第一个装配站的装配时间。因此装配站问题的边界条件就是:

$$s[1,1] = k[1,1] + a[1,1]$$

$$s[2,1] = k[2,2] + a[2,2]$$

背包问题的边界条件很简单，就是没有装入任何物品的状态：

$$s[0,V\text{max}] = 0$$

确定最长公共子序列问题的边界条件，要从其决策方式入手，当两个字符串中的一个长度为 0 的时候，其公共子序列长度肯定是 0，因此其边界条件就是：

$$s[i,j] = 0; \qquad i=0 \text{ 或 } j=0$$

3.3.2 动态规划法的例子：字符串的编辑距离

我们把两个字符串的相似度定义为：将一个字符串转换成另外一个字符串时需要付出的代价。转换可以采用插入、删除和替换三种编辑方式，因此转换的代价就是对字符串的编辑次数。字符串转换的方法不唯一，以字符串"SNOWY"和"SUNNY"为例，下面是两种将"SNOWY"转换成"SUNNY"的方法。

- 转换方法 1：

```
S - N O W Y
S U N N - Y
```

转换代价 Cost = 3（插入 U、替换 O、删除 W）

- 转换方法 2：

```
- S N O W - Y
S U N - - N Y
```

转换代价 Cost = 5（插入 S、替换 S、删除 O、删除 W、插入 N）

不同的转换方法需要的编辑次数也不一样，最少的那个编辑次数就是字符串的编辑距离（edit distance）。

作为对比，首先给出一个朴素的递归算法，递归算法一如既往地简单优雅：

```
int EditDistance(char *src, char *dest)
```

```
{
    if((strlen(src) == 0) || (strlen(dest) == 0))
        return abs(strlen(src) - strlen(dest));

    if(src[0] == dest[0])
        return EditDistance(src + 1, dest + 1);

    int edIns = EditDistance(src, dest + 1) + 1; //source 插入字符
    int edDel = EditDistance(src + 1, dest) + 1; //source 删除字符
    int edRep = EditDistance(src + 1, dest + 1) + 1; //source 替换字符

    return min(min(edIns, edDel), edRep);
}
```

显然，朴素的递归算法时间复杂度是 $O(3^n)$，对于以上两个长度为 5 的字符串，递归调用的次数是 241 次，接近于 3^5 这个量级，当字符串的长度非常大的时候，这个算法将变得不能接受。

现在考虑用动态规划的方法对这个算法进行改进，这个问题的阶段划分不是很明显，我们首先定义出问题的状态，从状态转换关系开始入手定义阶段和子问题的递推关系。假设 source 字符串有 n 个字符，target 字符串有 m 个字符，如果将问题定义为求解将 source 的[1…n]个字符转换为 target 的[1…m]个字符所需要的最少编辑次数（编辑距离），则其子问题就可以定义为将 source 的前[1…i]个字符转换为 target 的[1…j]个字符所需要的最少编辑次数，这就是本问题的最优子结构，因此我们将状态 $d[i,j]$定义为从子串 source[1…i]到子串 target[1…j]之间的编辑距离。

根据状态的定义，两个长度是 5 的字符串最多可以有 25 个状态，朴素递归方法之所以递归的次数达到了 3^5 数量级，就是因为大量的状态是重复计算的，没有剪枝优化。现在采用动态规划的思想对朴素递归算法进行改造，首先引入状态的概念，递归接口增加状态标志参数 i 和 j，其次是引入备忘录概念，用一个二维表记录每个状态的值，递归过程中优先进行查表。所有的状态记录在一个二维表中，二维表的每个元素定义如下：

```
typedef struct tagMemoRecord
{
    int distance;
    int refCount;
}MEMO_RECORD;
```

其中 distance 是编辑距离，初始化为 0xFFFF，表示一个无效的状态。refCount 是状态记录被引用的次数，0 表示没有这个状态的记录。调整后的算法如下：

```
int EditDistance(char *src, char *dest, int i, int j)
{
    if(memo[i][j].refCount != 0) //查表，直接返回
    {
        memo[i][j].refCount++;
        return memo[i][j].distance;
    }
    int distance = 0;
    if(strlen(src + i) == 0)
    {
```

```
            distance = strlen(dest + j);
        }
        else if(strlen(dest + j) == 0)
        {
            distance = strlen(src + i);
        }
        else
        {
            if(src[i] == dest[j])
            {
                distance = EditDistance(src, dest, i + 1, j + 1);
            }
            else
            {
                int edIns = EditDistance(src, dest, i, j + 1) + 1; //插入字符
                int edDel = EditDistance(src, dest, i+1, j) + 1; //删除字符
                int edRep = EditDistance(src, dest, i+1, j+1) + 1; //替换字符
                distance = min(min(edIns, edDel), edRep);
            }
        }

        memo[i][j].distance = distance;
        memo[i][j].refCount = 1;

        return distance;
    }
```


仍以前文提到的两个字符串为例，采用动态规划的递归方案，递归调用的次数减少到 40 次。25 个状态中，有很多状态被引用的次数都超过了 2 次，最多的达到了 3 次，说明通过查表有效地减少了递归搜索的次数，这就是前文提到的动态规划法内在的剪枝机制。如果不考虑状态记忆表的查询和维护开销，算法的时间复杂度已经接近 $O(n^2)$的级别。

现在已经定义了状态，并且 EditDistance()函数中也体现了状态转换关系和状态的边界条件，接下来我们可以直接给出状态递推关系方式的动态规划算法。根据决策方式，$d[i, j]$的递推关系分为两种情况，分别是 source[i]等于 target[j]和 source[i]不等于 target[j]，两种情况下 $d[i, j]$的递推关系如下：

$d[i,j] = d[i-1,j-1] + 0$; source[i]等于 target[j]

$d[i,j] = \min(d[i,j-1]+1，d[i-1,j]+1，d[i-1,j-1]+1)$; source[i]不等于 target[j]

当 target 字符串是空字符串时，编辑距离相当于将 source 字符串中的字符逐个删除的次数，因此可以确定一个边界条件为：

$d[i,0]$ = source 字符串的长度

同样，如果 source 字符串的长度为 0，编辑距离相当于在 source 字符串中逐个插入 target 字符的次数，因此另一个边界条件就是：

$d[0,j]$ = target 字符串的长度

确定了递推关系和边界条件，就可以给出直接利用状态递推关系实现的动态规划算法：

```
int EditDistance(char *src, char *dest)
{
    int i,j;
    int d[MAX_STRING_LEN][MAX_STRING_LEN] = { 0xFFFF };

    for(i = 0; i <= strlen(src); i++)
        d[i][0] = i;
    for(j = 0; j <= strlen(dest); j++)
        d[0][j] = j;

    for(i = 1; i <= strlen(src); i++)
    {
        for(j = 1; j <= strlen(dest); j++)
        {
            if((src[i - 1] == dest[j - 1]))
            {
                d[i][j] = d[i - 1][j - 1]; //不需要编辑操作
            }
            else
            {
                int edIns = d[i][j - 1] + 1; //source 插入字符
                int edDel = d[i - 1][j] + 1; //source 删除字符
                int edRep = d[i - 1][j - 1] + 1; //source 替换字符

                d[i][j] = min(min(edIns, edDel), edRep);
            }
        }
    }

    return d[strlen(src)][strlen(dest)];
}
```

以上是一个动态规划法应用的实例，从朴素的递归方式开始，通过明确状态定义，逐步过渡到动态规划法实现，帮助大家体会动态规划的设计思想。虽然动态规划的概念很抽象，但是只要确定了问题的实质，按照 3.3.1 节给出的四个步骤逐步分析，实现动态规划法的算法也不是一件很困难的事情。有时候，如果直接用递推方式很难写出的算法实现，不妨考虑采用带备忘录的递归方式，谁说这不是动态规划？

3.4 解空间的穷举搜索

不要误会，本节要介绍的就是*穷举法*（穷举搜索法）。解空间又称为状态空间，是所有可能是解的候选解的集合，之所以特别强调在解空间内穷举搜索，是想传达一个重要的思想，那就是穷举并不是漫无目的地乱找，它是一种在有限的解空间（解空间至少在理论上是有限的）内按照一定的策略进行查找的思想。数学上也把穷举法称为枚举法，就是在一个由有限个元素构成的集合中，将所有元素一一枚举研究的方法。比如要找出一个班上身高最高的同学，只需要给这个班上的同学一一测量身高，然后通过比较就可以确定哪个同学身高最高。穷举法就是这样一种思想，

对解空间内的候选解按某种顺序进行逐一枚举和检验，并根据问题给定的条件从中找出那些符合要求的候选解作为问题的解。穷举法一般可以找出解空间中所有正确的解，如果给定最优解的判断条件，穷举法也可以用于求解最优解问题。

一般来说，只要一个问题有其他更好的解决方法，通常不会选择穷举法，穷举法也常被作为“不是办法的办法”或“最后的办法”。但是绝对不能因为这样而轻视穷举法，穷举法在算法设计模式中占有非常重要的地位，它还是很多问题的唯一解决方法。穷举法虽然思想简单，但是设计一个解决特定问题的穷举法实现却并不简单。首先，解空间或状态空间的定义没有具体的模式，不同问题的解空间形式上差异巨大。其次，针对不同问题都要选择不同的搜索算法，有很多问题的搜索算法并不直观，需要对问题做细致的分析才能得到。正因为如此，穷举法也被公认为是最“难用”的算法模式，看起来简单，但是面对问题往往无从下手。但是如果能用好穷举法，你就掌握了能解决所有问题的“通用算法”，至少理论上是这样。

穷举法的基本思想就是以下两个步骤。

(1) 确定问题的解（或状态）的定义，解空间的范围以及正确解的判定条件。

(2) 根据解空间的特点选择搜索策略，一一检验解空间中的候选解是否正确，必要时可辅助一些剪枝算法，排除一些明显不可能是正确解的检验过程，提高穷举的效率。

正如前面所讲的那样，穷举法的设计思想非常简单，没有任何条件性的约束和假设，使得穷举法几乎适合求解任何问题，当然，穷举法的“难用”也体现在这两个步骤上。

3.4.1 解空间的定义

解空间就是全部可能的候选解的一个约束范围，确定问题的解就在这个范围内，将搜索策略应用到这个约束范围就可以找到问题的解。用“空间”这个词是为了说明候选解不一定是线性结构，根据问题的类型，解空间的结构可能是线性表、集合、树或者图。有时候，这个空间内的对象不是问题的解，而是一些被称为状态的对象，通过对状态的计算和处理，可以间接地得到问题的解，这样的搜索空间也常被理解为状态空间。

要确定解空间，首先要定义问题的解，建立解的数学模型。如果解的数学模型选择错误或不合适，会导致解空间结构繁杂，范围难以界定，甚至无法设计搜索算法。以 3.1.2 节给出的 0-1 背包问题为例，如果将物品的最大价值定为解的数学模型，则解空间内的候选解就是某几件物品的价值总和，解空间的范围就是[0, 235]，235 是全部 7 件物品的价值总和。如果对这个解空间穷举搜索，就需要根据每一个价值总和反推出这个价值总和由哪几个物品组成，这会使搜索算法非常麻烦。如果换一个角度考虑这个问题，将解的数学模型定义为物品的选择状态，用一个 7 元组分别表示 7 件物品的选择状态，0 表示不选择装入该物品，1 表示选择装入该物品。根据之前解题的答案，最优解是选择 1、2、4、6、7 号物品，用 7 元组表示就是[1,1,0,1,0,1,1]。根据这个选择状态，计算最终的物品总价值的方法非常简单，直接求和即可，比前一种方案的根据价值总和反推物品选择状态也简单很多。根据状态定义，解空间一共有 128（2^7）个状态，非法解判断与合法解的判断，以及最优解的比较算法都非常简单。最重要的是，搜索算法的设计也很简单，n

元组的遍历有递归、多重循环等多种成熟的实现方法可以选择，简单套用即可。

上例中的解空间是一种相对简单的定义，候选解或状态之间相互独立，没有关联关系，可以用线性表，也可以用集合来组织解空间。在很多情况下，候选解或状态之间不独立，存在各种关联关系，第 6 章介绍的“妖怪与和尚过河问题”就是一个这样的例子。妖怪与和尚用一个 6 元组定义状态，0 表示在河左岸，1 表示在河右岸，初始状态是[0,0,0,0,0,0]，最终解的状态是[1,1,1,1,1,1]。这些状态之间没有简单的规律，不能用一套通用的遍历算法将这些状态都事先确定好，但是可以根据状态之间的演化关系，从一种状态推出另一种或几种状态，递归地执行这种状态演化，逐步得到整个状态空间。在这种情况下，解空间通常伴随着搜索算法展开，从一个原始状态开始，逐步扩展至整个解空间。这样的解空间通常被组织成一棵状态树，最终状态就是状态树的叶子节点，从根节点到叶子节点之间的状态转换过程就是问题求解的过程。对于更复杂的情况，需要用图的一些方法组织和搜索解空间，在这种情况下，解空间就是节点和边的关系空间。

3.4.2 穷举解空间的策略

穷举解空间的策略就是搜索算法的设计策略，简单的问题可以用通用的搜索算法，比如 0-1 背包问题的解空间可以用排列组合算法得到，复杂的问题需要根据实际情况设计搜索算法。根据问题的需要设计搜索算法是一件困难重重的事情，没有捷径，只能在常用搜索策略的基础上多实践，多积累。盲目搜索和启发性搜索是两种最常用的搜索策略。顾名思义，*盲目搜索*就是不带任何假设的穷举搜索，不管行不行，眉毛胡子一把抓。*启发性搜索*是利用某种策略或计算依据，有目的地搜索，这些策略和依据通常能够加快算法的收敛速度，或者能够划定一个更小的、最有可能出现解的空间并在此空间上搜索。

一般来说，为了加快算法的求解，通常会在搜索算法的执行过程中伴随一些剪枝动作。剪枝是一个很形象的比喻，如果某一个状态节点确定不可能演化出结果，就应该停止从这个状态节点开始的搜索，相当于状态树上这一分枝就被剪掉了。除了采用剪枝策略，还可以使用限制搜索深度的方法加快算法的收敛，但是限制搜索深度会导致无解，或错过最优解，通常只在特定的情况下使用，比如博弈树的搜索。

1. 盲目搜索算法

广度优先搜索和深度优先搜索是两种常用的盲目搜索算法，这种搜索算法只根据问题的规模，按照广度优先和深度优先的原则搜索解空间内的每一个状态。广度优先和深度优先的算法模式已经在第 2 章介绍过，这里不再赘述。广度优先算法因为需要额外的存储空间，因此在设计算法时要考虑此额外空间的规模。深度优先算法在搜索过程中容易陷入状态循环，导致在一个没有解的子树上“死循环”，一般需要做状态循环的判断和避免。但总的来说，两种策略并无优劣之分，很多情况下可以互换使用。

2. 启发式搜索算法

很多情况下，当问题的规模达到一定的程度，盲目搜索算法就会因为低效而被排斥，理论上

可以得到答案，但是要等一万年，这是人类不能接受的结果。如果搜索能够智能化一点，利用搜索过程中出现的额外信息直接跳过一些状态，避免盲目的、机械式的搜索，就可以加快搜索算法的收敛，这就是启发性搜索。启发性搜索需要一些额外信息和操作来“启发”搜索算法，根据这些信息的不同，启发方式也不同。如果知道解空间的状态分布呈现正态分布的特征，如图 3-1 所示，则可以从分布中间值开始向两边搜索，因为在中间值附近出现最优解的概率更高，这就是启发式搜索。如果能有一个状态评估函数，可以对每个状态节点能演化出解的可能性进行评估，搜索过程中根据这种可能性对待搜索的状态节点排序，也是一种启发式搜索。再简单一点，如果在某一个层面的搜索能应用贪婪策略，优先选择与贪婪策略符合的状态节点进行搜索，也是一种启发式搜索。第 21 章介绍的 A*寻径算法，也是一种带启发的搜索算法，利用路径评估函数，每次都选择距离出发点最近的位置开始搜索。

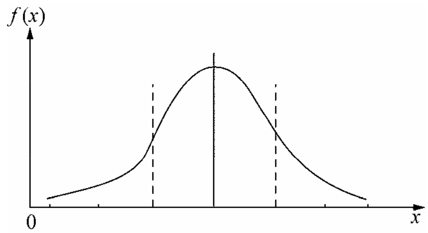


图 3-1 正态分布示意图

3. 剪枝策略

对解空间穷举搜索时，如果有一些状态节点可以根据问题提供的信息明确地被判定为不可能演化出最优解，也就是说，从此节点开始遍历得到的子树，可能存在正确的解，但是肯定不是最优解，就可以跳过此状态节点的遍历，这将极大地提高算法的执行效率。这就是剪枝策略。应用剪枝策略的难点在于如何找到一个评价方法（估值函数）对状态节点进行评估。特定的评价方法都附着在特定的搜索算法中，比如博弈树算法中常用的极大极小值算法和“α - β”算法，都伴随着相应的剪枝算法。除了针对特定问题类型的剪枝算法之外，没有可以一统天下的通用评价方法，通常需要根据实际问题小心地分析，确定评价方法。

除了最优解问题，还有一种情况也会用到剪枝策略。对解空间内的状态节点遍历搜索的过程中，会有一些在特定搜索策略下重复出现的状态节点，对这些状态节点如果不做特殊处理，不仅会因为重复处理相同的状态节点而降低效率，还可能会导致深度优先搜索算法“陷入”到某个子树的搜索中无法退出。举个例子，如果出现对状态 A 搜索得到子状态 B，对状态 B 搜索得到子状态 C，对状态 C 搜索又可得到子状态 A 的情况，就会使得搜索算法陷入“死循环”。在这种情况下，常用的剪枝策略就是找到一种算法对状态计算校验值，通过比较校验值判断是否是已经处理过的状态节点。第 22 章介绍华容道游戏的自动求解算法时，就用到了这种剪枝策略。

4. 搜索算法的评估和收敛

穷举法虽然被称为灵活的“通用算法”，但也不是万能的，穷举法最大的敌人是问题的规模。很多问题，当规模大到一定程度时，使用穷举法就只具有理论上的可行性。对某些问题，穷举法是最后的办法，但是问题规模又大到无法对解空间进行完整的搜索，这时候就需要对搜索算法进行评估，并确定一些收敛原则。收敛原则就是只要能找到一个比较好的解就返回（不求最好），根据解的评估判断是否需要继续下一次搜索。大型棋类游戏通常面临这种问题，比如国际象棋和

围棋的求解算法，想要搜索整个解空间得到最优解目前是不可能的，所以此类搜索算法通常都通过一个搜索深度参数来控制搜索算法的收敛，当搜索到指定的深度时（相当于走了若干步棋）就返回当前已经找到的最好的结果，这种退而求其次的策略也是不得已而为之，第 23 章介绍博弈树和棋类游戏的时候，会具体介绍相关的方法。

3.4.3 穷举搜索的例子：Google 方程式

有一个由字符组成的等式：WWWDOT - GOOGLE = DOTCOM，每个字符代表一个 0 ~ 9 之间的数字，WWWDOT、GOOGLE 和 DOTCOM 都是合法的数字，不能以 0 开头。请找出一组字符和数字的对应关系，使它们互相替换，并且替换后的数字能够满足等式。据说这是 Google 公司的面试题，我没有考证过，不过这种字符方程（或字符等式）问题有很多变种，比如 2005 年的 Google 中国编程挑战赛第二轮淘汰赛有一道名为“SecretSum”的 500 分的竞赛题，与本题如出一辙，只不过字母是 3 个，而且用的是加法计算。这个问题其实并不难，你可以将其列成竖式减法的形态，然后人工推算出来，不过接下来我们要使用穷举法来求解这个问题。

从穷举法的角度看，这是一个典型的排列组合问题，题目中一种出现了 9 个字母，每个字母都可能是 0 ~ 9 之间的数字，穷举的方法就是对每个字母用 0 ~ 9 的数字尝试 10 次，如果某一次得到的字母和数字的对应关系能够满足减法等式，则输出这一组对应关系。根据题目意思，每个字母代表一个数字，也就是说，如果 W 代表 1，则其他 8 个字母就不可能是 1。很显然，这是个组合问题，如果不考虑 0 开头数字的情况，这样的组合应该有 $10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2=3628800$ 种组合，在这样的数量级上使用穷举法，计算机处理起来应该没有压力。

现在考虑给出一种解决这种字符方程问题的通用解法。从数据结构定义上，首先要避免使用固定 9 个字符的方法，这就需要定义一个可变化的字符元素列表，每个字符元素包含 3 个属性，分别是字母本身、字母代表的数字以及是否是数字的最高位（根据题意，最高位不能是 0，所以要特别对待）：

```
typedef struct tagCharItem
{
    char c;
    int value;
    bool leading;
}CHAR_ITEM;
```

对于本题，这个列表可以初始化为：

```
CHAR_ITEM charItem[] = { { 'W', -1, true  }, { 'D', -1, true  }, { 'O', -1, false },
                         { 'T', -1, false }, { 'G', -1, true  }, { 'L', -1, false },
                         { 'E', -1, false }, { 'C', -1, false }, { 'M', -1, false } };
```

如果换成 Google 编程挑战赛的“SecretSum”题目，这个列表可以初始化为：

```
CHAR_ITEM charItem[] = { {'A', -1, true}, {'B', -1, true}, {'C', -1, true},
                         {'D', -1, false} };
```

因为这是一个组合问题，两个字母不能被指定为相同的数字，这就需要对每个数字做一个标识，当这个数字已经被某个字符“占用”时，其他字符不能再使用这个数字。我们对可参与穷举的数字也定义一个列表，对于这个问题来说，0 ~ 9 都可以参与穷举，但是有的问题可能有特殊的约束，比如字符只能代表偶数，或只能代表奇数等，每个数字元素有一个额外的占用标识：

```
typedef struct tagCharValue
{
    bool used;
    int value;
}CHAR_VALUE;
```


穷举的搜索算法采用递归的方式进行组合枚举，按照 charItem 列表中的顺序，逐个对每个字符进行数字遍历，算法实现如下：

```
void SearchingResult(CHAR_ITEM ci[], CHAR_VALUE cv[],
    int index, CharListReadyFuncPtr callback)
{
    if(index == max_char_count)
    {
        callback(ci);
        return;
    }

    for(int i = 0; i < max_number_count; ++i)
    {
        if(IsValueValid(ci[index], cv[i]))
        {
            cv[i].used = true;/*set used sign*/
            ci[index].value = cv[i].value;
            SearchingResult(ci, cv, index + 1, callback);
            cv[i].used = false;/*clear used sign*/
        }
    }
}
```

根据题目要求，W、G 和 D 这 3 个字符不能是 0，因此枚举过程中对这 3 个字符是 0 的情况进行剪枝，IsValueValid()函数就是评估函数，通过剪枝操作，callback 被调用的次数由理论上的 3628800 次减少为 2540160，减少了约 30%的计算判断。index 参数标识字符索引，每次递归调用对索引为 index 的字符进行数字遍历，当 index 等于字符个数时，表示所有的字符都已经指定了对应的数字，可以调用 callback 进行结果判断。SearchingResult()函数可以作为此类问题的通用搜索框架，只需指定不同的 CHAR_ITEM 和 CHAR_VALUE 参数，以及结果处理回调 callback 即可。对于本题，callback 函数可以这样编写：

```
void OnCharListReady(CHAR_ITEM ci[])
{
    char *minuend    = "WWWDOT";
    char *subtrahend = "GOOGLE";
    char *diff       = "DOTCOM";

    int m = MakeIntegerValue(ci, minuend);
```

```
    int s = MakeIntegerValue(ci, subtrahend);
    int d = MakeIntegerValue(ci, diff);
    if((m - s) == d)
    {
        std::cout << m << " - " << s << " = " << d << std::endl;
    }
}
```

对整个解空间搜索以后，只得到以下两个合法的答案：

```
777589 - 188103 = 589486
777589 - 188106 = 589483
```

3.5　总结

本章介绍了几种设计算法常用的思想，这些方法之间既有相同点，也有差别。模式作为算法演进的一些固定的思路，提供了一些构造算法的常用思想，但不是构造算法的全部方法，不可以将其作为特定的框架套用，而应该具体问题具体分析，在了解各种算法思想的原理和适用原则的基础上，灵活地运用这些算法设计思想。本书随后章节给出的各种趣味算法以及这些算法在现实中的应用，处处都可见这些算法设计思想的影子。

3.6　参考资料

[1] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[2] 维基百科：http://zh.wikipedia.org/wiki/算法模式

[3] 维基百科：http://zh.wikipedia.org/wiki/动态规划

[4] Levitin A. 算法设计与分析基础. 潘彦译. 北京：清华大学出版社，2007

[5] Kleigberg J, Tardos E. *Algorithm Design*. Addison-Wesley, 2005

[6] Sahni S. 数据结构算法与应用——C++语言描述（英文版）. 北京：机械工业出版社，2007

[7] 维基百科：http://zh.wikipedia.org/wiki/分治法

[8] Karatsuba A,Ofman Yu. *Multiplication of Many-Digital Numbers by Automatic Computers*. Proceedings of the USSR Academy of Sciences, 1962

[9] Knuth D E. *The Art of Computer Programming*（*Third Edition*）,Vol 2. Addison-Wesley, 1997

[10] 维基百科“Karatsuba 算法”：http://en.wikipedia.org/wiki/Karatsuba_algorithm

[11] Sedgewick R. *Algorithm in C*. Addison-Wesley,2001

[12] 刘汝佳，黄亮. 算法艺术和信息学竞赛. 北京：清华大学出版社，2003

第4章 阿拉伯数字与中文数字

在人类文明的进化过程中，数字很可能早于文字出现。不同民族的计数方法也是千奇百怪的，例如，在英语中表示数字的单词是 digit，而在其祖先拉丁语中，digitus 表示手指，digitos 表示脚趾，根据它们共有的词根 digit，不难猜想他们最初的计数工具是什么。也有一些民族采用小石子或通过在绳子上打结来计数，据说非洲有些部落用鳄鱼计数，可见部落民风之彪悍。据说，因纽特人不擅长计数，他们曾与世隔绝生存了 4000 多年，因此让因纽特人从 1 数到 6 都需要相当大的勇气。最早到来的白人殖民者居然让因纽特人每天统计狩猎情况，结果导致了因纽特人的暴动。看来数数还是个性命攸关的大事，这些白人殖民者不仅缺德，还缺心眼儿。

掌握计数方法是人类文明的基本要素之一，很多古代文明都有自己的数字体系，比如古埃及和印度使用的十进制数字、玛雅人使用的二十进制数字，等等。中文数字记数方式以及记账的大写数字是中国特有的数字体系，用方块字描述数字，用符合中文语法的方式书写、记录数字，具有鲜明的中国特色。中文数字的计数方式与全球通用的阿拉伯数字体系的计数方式迥然不同，两种计数方式之间的转换就成为一个很有意思的话题。这个主题也很适合作为程序员招聘的面试题，网上有很多类似的转换算法，但是大部分都有问题。本章将介绍中文数字与阿拉伯数字相互转换的算法，并给出一个中文数字转换的测试用例，该用例符合国家对中文数字使用的相关标准，也符合本章最后给出的两个行业标准的规定。经过测试，网上能找到的公开的算法基本上都不满足该测试用例。

4.1 中文数字的特点

中文数字也采用十进制，用汉字“零一二三四五六七八九”表示基本记数，与阿拉伯数字靠数字偏移位置暗示数字的权位不一样，中文数字直接用“数字+权位”的方式组成数字，比如阿拉伯数字 100，用中文计数就是“一百”，其中“一”是数字，“百”是数字的权位。中文常用的数字权位有“十”“百”“千”“万”“亿”等，这些权位对应的阿拉伯数字记数单位（偏移位置）如下：

十　10

百　100

千　1000

万　10000

亿　100000000

除了以上几种权位，中国古代还有兆、京、垓等表示更大尺度的数字权位，但是关于这些数字权位的定义却各不相同，有的计数方式是万万为亿，亿亿为兆，兆兆为京，还有的计数方式是万万为亿、万亿为兆、万兆为京。现代科学技术中常说的“兆”已经被定义为一百万，因此现代中文计数不再使用“兆”和比“兆”更大的单位，而是用“万亿”、“百万亿”、“千万亿”、“亿亿”来堆叠计数。

4.1.1　中文数字的权位和小节

中文数字的特点之一就是每个计数数字都跟着一个权位，这个权位就是这个数字的量值，相当于阿拉伯数字中的数位。比如中文数字“一千二百三十”，数字“一”的权位是“千”，对应的阿拉伯数字的值是 1×1000，数字“二”的权位是“百”，对应的阿拉伯数字的值是 2×100，数字“三”的权位是“十”，对应的阿拉伯数字的值是 3×10，整个数字的值就是 $1\times1000+2\times100+3\times10=1230$。最低位数字没有权位，也可以理解为权位是空。

中文计数的另一个特点是以“万”为小节（欧美的计数习惯是以“千”为小节），每一个小节都有一个节权位，万以下的节没有节权位（或节权位是空），万以上的节权位就是万，再大就是亿（即万的一万倍）。每个小节内部以“十百千”为权位独立计数。“十百千”这几个权位是不能连续出现的，比如“二十百”和“一千千”都是非法的中文数字，但是“万”和“亿”作为节权位时可以和其他权位连在一起使用，比如“二十亿”和“五千万”都是合法的中文数字。

4.1.2　中文数字的零

中文计数还有一个特点，就是“零”的使用变化多端。阿拉伯数字中数字的权位依靠数字在整个数字长度中的偏移位置确定，因此数字中间出现的 0 用于标记数字的偏移位置，即便是连续出现的 0 也不能省略。中文计数方式中每个数字的权位都直接跟在数字后面，因此可以用一个“零”代表连续出现的若干个 0。尽管如此，也不是所有的情况都使用“零”，比如阿拉伯数字 20001234，中文数字表示为“二千万一千二百三十四”，没有用一个“零”；再比如阿拉伯数字 12000，中文数字表示为“一万二千”，也没有用“零”；但是对于阿拉伯数字 10210300，中文数字表示为“一千零二十一万零三百”，两次出现“零”。

中文数字对“零”的使用总结起来有以下三条规则。

- 规则 1：以 10000 为小节，小节的结尾即使是 0，也不使用“零”。
- 规则 2：小节内两个非 0 数字之间要使用“零”。
- 规则 3：当小节的“千”位是 0 时，若本小节的前一小节无其他数字，则不用“零”，否则就要用“零”。

4.2 阿拉伯数字转中文数字

阿拉伯数字与中文数字没有一一对应关系，不存在直接转换的公式化算法，因此需要根据两种数字体系的特点精心构造转换算法。从阿拉伯数字到中文数字的转换，第一步是以“万”为单位分节，并确定节权位。第二步是对每小节内的数字确定权位，并按照本章 4.1.2 节给出的三条规则处理“零”的问题。

4.2.1 一个转换示例

以阿拉伯数字 200001010200 为例，首先以“万”为单位对其分节，可分为三节：2000 0101 0200，第一节 2000，节权位是“亿”，因为这一节的 0 都在结尾，根据规则 1，此处不使用“零”，直接表示为“二千亿”。第二节 0101，节权位是“万”，因两个 1 之间有 0，根据规则 2，101 可以描述为“一百零一”。另外，此节的千位是 0，根据规则 3，因本小节前还有数字，因此需要用“零”。也就是说，本小节需要两个“零”。最后一个小节，结尾的两个 0 根据规则 1，不使用“零”，但是千位的 0 根据规则 3，需要使用“零”。根据以上分析，将三个小节的转换结果组合在一起，阿拉伯数字 200001010200 的中文表示就是“二千亿零一百零一万零二百”。

从这个例子可以看出来，对阿拉伯数字分节，确定数字的权位很简单，最难处理的就是 0 的转换，需要根据三个规则灵活选择是否需要使用“零”。

4.2.2 转换算法设计

设计阿拉伯数字转中文数字的算法，也可以遵循上例中的两个步骤来处理，但是需要解决三个问题。第一个问题是单个数字的转换，这个并不难，因为阿拉伯数字 0 ~ 9 与相应的中文数字是一一对应的。对这个转换设计算法非常简单，可以利用第 2 章介绍的数组下标的技巧，这样定义中文数字表：

```
const char *chnNumChar[CHN_NUM_CHAR_COUNT] = { "零", "一", "二", "三", "四", "五", "六", "七",
    "八", "九" };
```

待转换的阿拉伯数字作为数组下标，比如 chnNumChar[5]就是阿拉伯数字 5 对应的中文数字。

第二个需要解决的问题是节与权位的识别。节的划分很简单，以“万”为单位截断即可。节权位的定义也采用一维表，可以利用数组下标直接定位出节权的中文名称：

```
const char *chnUnitSection[] = { "", "万", "亿", "万亿" };
```

对于 32 位正数能表达的最大数来说，最大节权是“万亿”已经足够了，如果要转换更大的数，可以延伸这个节权表的定义，比如增加“亿亿”。数字中最低的节没有节权，使用空字符串作为占位符也是一个算法设计常用的一致性处理的技巧：对最低的节不做特殊处理，和其他节一样指定节权位，只不过节权位是空字符串，对转换出的中文数字最终结果没有影响。每个节内的数字对应的权位也采用这种方式定义：

```
const char *chnUnitChar[] = { "", "十", "百", "千" };
```

最低位的权位是空字符串，处理方式和节权位的处理方式一样。数字权位的确定并不困难，通过移位就可以确定每个数字对应的权位。阿拉伯数字的权位是隐含在数字的位数中的，使用 0 作为占位符。比如数字 1000，要使 1 处在千位，一定会补 3 个 0 作为占位符，否则 1 就不代表“一千”。既然每一位的权都在固定的位置上，只要记录移位的次数就可以确定阿拉伯数字的权位，以移位次数做下标，直接查 chnUnitSection 和 chnUnitChar 表就可以得到正确的中文数字的权位。

第三个需要解决的问题是如何处理中文“零”。这个问题稍微有点困难，需要根据 4.1.2 节的三个规则灵活判断，此外，对于连续出现的阿拉伯数字 0，也只能用一个中文“零”。

4.2.3　算法实现

转换算法首先要对阿拉伯数字分节，并确定节权位名称。num 对 10000 取模可得到一个 section，将这个 section 转成中文数字，然后根据节的位置补上节权位，即可完成一个节的中文数字转换。重复这个过程，直到 num 等于 0 为止，整个转换就算完成。unitPos 变量记录节的位置，0 对应空字符串，1 对应“万”，2 对应“亿”，随着 unitPos 的增加，节权位也越来越大。全 0 的节不需要节权位，这个在代码中也有处理。根据 4.1.2 节规则 3 的定义，如果一节内数字的千位是 0，需要根据前面是否还有数字决定是否需要加“零”，NumberToChinese()函数中利用变量 needZero 和 while(num > 0)循环语句，巧妙地做了这个加“零”处理，省去了一个 if 判断。

```
//num == 0 需要特殊处理，直接返回"零"
void NumberToChinese(unsigned int num, std::string& chnStr)
{
    int unitPos = 0;
    std::string strIns;
    bool needZero = false;

    while(num > 0)
    {
        unsigned int section = num % 10000;
        if(needZero)
        {
            chnStr.insert(0, chnNumChar[0]);
        }
        SectionToChinese(section, strIns);
        /*是否需要节权位? */
        strIns += (section != 0) ? chnUnitSection[unitPos] : chnUnitSection[0];
        chnStr.insert(0, strIns);
        /*千位是 0 需要在下一个 section 补零*/
        needZero = (section < 1000) && (section > 0);
        num = num / 10000;
        unitPos++;
    }
}
```

SectionToChinese()函数将一个节的数字转换成中文数字，利用中文数字表 chnNumChar 转换

中文数字，利用表 chnUnitChar 得到数字权位，unitPos 变量用作权位索引。SectionToChinese() 函数的关键部分是对 0 的处理，根据规则 1 和规则 2，小节结尾的 0 不需要转换成“零”，但是两个数字之间的 0 需要转换成“零”。如果两个数字之间有多个 0，也只转换一个“零”，变量 zero 用于控制“零”的转换，避免出现多个“零”连在一起的情况。

```
void SectionToChinese(unsigned int section, std::string& chnStr)
{
    std::string strIns;
    int unitPos = 0;
    bool zero = true;
    while(section > 0)
    {
        int v = section % 10;
        if(v == 0)
        {
            if(!zero)
            {
                zero = true; /*需要补，zero 的作用是确保对连续的多个，只补一个中文零*/
                chnStr.insert(0, chnNumChar[v]);
            }
        }
        else
        {
            zero = false; //至少有一个数字不是
            strIns = chnNumChar[v]; //此位对应的中文数字
            strIns += chnUnitChar[unitPos]; //此位对应的中文权位
            chnStr.insert(0, strIns);
        }
        unitPos++; //移位
        section = section / 10;
    }
}
```

4.2.4 中文大写数字

中文数字还有一个很有意思的现象，就是中文数字大写。所谓的大写其实就是用一些笔画复杂的汉字代替简单的数字汉字，其目的就是为了保证其不容易被篡改。中文大写用“壹贰叁肆伍陆柒捌玖”代替“一二三四五六七八九”，用“拾佰仟”代替“十百千”。这些数字的繁写其实在唐代就已经出现，但正式作为记载钱粮、税收等项目用的官方数字，是在明朝初年著名的“郭桓案[①]”之后。

实现中文大写数字的转换，只需要将 chnNumChar、chnUnitSection 中的中文数字和权位名称

① 郭桓案：与空印案、胡惟庸案和蓝玉案一起并称为明初四大案。郭桓案发生在明朝洪武十八年（1385 年），属于官吏贪污案件。户部侍郎郭桓等人，串通地方官吏作弊，篡改账册，私吞太平、镇江等府的赋税，还盗卖官粮。后被揭发，以其涉案金额巨大，对经济领域影响深远而为世人瞩目，对此，明太祖将六部左、右侍郎以下官员全部处死，地方官吏死于狱中者达数万人以上。为了追赃，牵连到全国各地的小富百姓，遭到抄家破产的不计其数。由于牵扯面过广，全国百姓对此案非常不满意，明太祖为了平息民怨，将审刑官吴庸等人也一并处死。

替换成大写数字就可以了，转换算法是一样的。如果用于人民币记账，可调整节权位的名称，加上“圆”或“圆整”等权名，有兴趣的读者可自行完成转换代码。

4.3 中文数字转阿拉伯数字

中文数字的权位是明的，阿拉伯数字的权位则隐含在数字的位置中。比如中文数字“一万”，对应的阿拉伯数字是 10000，如何确定补多少个 0 才能将 1 顶在正确的位置上？这正是中文数字转换成阿拉伯数字的关键，如何将明的权位转换成数字的位置。

4.3.1 转换的基本方法

对于十进制阿拉伯数字，数字的所在位数就是该数字与 10 的倍数关系。个位就是 1 倍，十位就是 10 倍，百位就是 100 倍，以此类推。通过这个关系，可以将阿拉伯数字隐含的权位转换成 10 的倍数表示，比如中文数字“五百”，就可以转换成 5×100，其结果就是 500。再来看一个复杂的中文数字“四万二千五百一十三”，对每个权位依次转换成倍数并求和：4×10000 + 2×1000 + 5×100 +1×10+3×1，就可以得到对应的阿拉伯数字 42513。

由以上分析可知，从中文数字转阿拉伯数字的基本方法就是从中文数字中逐个识别出数字和权位的组合，然后根据权位和阿拉伯数字倍数的对应关系计算出每个数字和权位组合的值，最后求和得到结果。但是中文数字并不是严格用“数字”+“权位”组合成的，“零”的使用就是个特例，它在数字中出现，却没有权位。除此之外，节权位也需要考虑，因为它常和其他权位连在一起使用，比如“二十万”中的“十”是数字权位，“万”是节权位。在设计算法时，由于“零”没有权位，因此对于中文数字中的“零”不需处理，直接跳过即可。节权位比较特殊，它不是与之相邻的数字的倍数，而是整个小节的倍数，因此转换过程中，需要临时保存每个节权位出现之前的小节的值。

4.3.2 算法实现

中文数字转换成阿拉伯数字的算法实现，首先要做两件事情，一件是将中文数字转换成阿拉伯数字，另一件事情就是将中文权位转换成 10 的倍数。中文数字转换成阿拉伯数字可以通过反查 chnNumChar 表实现。将中文权位转成 10 的倍数需要事先建立一个中文权位与 10 的倍数的关系表，我们这样定义一个中文权位和 10 的倍数关系：

```
typedef struct
{
    const char *name; //中文权位名称
    int value; //10 的倍数值
    bool secUnit; //是否是节权位
}CHN_NAME_VALUE;
```

根据这个关系的定义建立的权位与 10 的倍数的关系表如下：

```
CHN_NAME_VALUE chnValuePair[] =
{
    { "十", 10, false }, { "百", 100, false }, { "千", 1000, false },
    { "万", 10000, true }, { "亿", 100000000, true }
};
```

根据以上定义实现的转换算法如下：

```
unsigned int ChineseToNumber(const std::string& chnString)
{
    unsigned int rtn = 0;
    unsigned int section = 0;
    int number = 0;
    bool secUnit = false;
    std::string::size_type pos = 0;

    while(pos < chnString.length())
    {
        int num = ChineseToValue(chnString.substr(pos, CHN_CHAR_LENGTH));
        if(num >= 0) /*数字还是单位？*/
        {
            number = num;
            pos += CHN_CHAR_LENGTH;
            if(pos >= chnString.length())//如果是最后一位数字，直接结束
            {
                section += number;
                rtn += section;
                break;
            }
        }
        else
        {
            int unit = ChineseToUnit(chnString.substr(pos, CHN_CHAR_LENGTH), secUnit);
            if(secUnit)//是节权位说明一个节已经结束
            {
                section = (section + number) * unit;
                rtn += section;
                section = 0;
            }
            else
            {
                section += (number * unit);
            }
            number = 0;
            pos += CHN_CHAR_LENGTH;
            if(pos >= chnString.length())
            {
                rtn += section;
                break;
            }
        }
    }

    return rtn;
}
```

ChineseToNumber()函数就是中文数字转阿拉伯数字算法的主要部分，chnString 参数就是合法的中文字符串，转换的过程就是对 chnString 中的中文逐个处理，如果遇到中文数字，就存放在 number 变量中，对于“零”不处理，直接跳过。如果是中文权位，则将其对应的倍数与 number 相乘得到对应的数字，同时累加到 section 变量中。如果是节权位，则将节权位对应的倍数与 section 相乘得到对应的数字，同时累加到最终的结果 rtn 变量中。ChineseToValue()函数负责查表完成中文数字到英文数字的转换，如果返回–1，则表示这是一个权位字符。ChineseToUnit()函数负责查 chnValuePair 表得到权位对应的 10 的倍数。

4.4 数字转换的测试用例

中文数字的表示方法随着地域的不同也有一些差异，比如数字 11，到底是“十一”还是“一十一”？再比如 110，到底是“一百一十”还是“一百一”？其实这些转换是有相应的国家标准和行业规定的。本书给出一套简单的测试用例，很多读者都写过自己的中文数字转换算法，不妨用这个测试用例检验一下，看看是否正确。

```
{0, "零"},
{1, "一"},
{2, "二"},
{3, "三"},
{4, "四"},
{5, "五"},
{6, "六"},
{7, "七"},
{8, "八"},
{9, "九"},
{10, "一十"},
{11, "一十一"},
{110, "一百一十"},
{111, "一百一十一"},
{100, "一百"},
{102, "一百零二"},
{1020, "一千零二十"},
{1001, "一千零一"},
{1015, "一千零一十五"},
{1000, "一千"},
{10000, "一万"},
{20010, "二万零一十"},
{20001, "二万零一"},
{100000, "一十万"},
{1000000, "一百万"},
{10000000, "一千万"},
{100000000, "一亿"},
{1000000000, "一十亿"},
{1000001000, "一十亿一千"},
{1000000100, "一十亿零一百"},
{200010, "二十万零一十"},
{2000105, "二百万零一百零五"},
{20001007, "二千万一千零七"},
```

```
{2000100190, "二十亿零一十万零一百九十"},
{1040010000, "一十亿四千零一万"},
{200012301, "二亿零一万二千三百零一"},
{2005010010, "二十亿零五百零一万零一十"},
{4009060200, "四十亿零九百零六万零二百"},
{4294967295, "四十二亿九千四百九十六万七千二百九十五"}
```

4.5 总结

中文数字体系有自己的特点，在财务和金融系统中有独特的用处。与轻巧的阿拉伯数字相比，在生活中使用和表达中文数字确实有不方便的地方，但是跟法语表达方式比起来，咱们中国人也没什么可抱怨的。不信你问问法国人，怎么用法语说 92 吧。

4.6 参考资料

[1]《中华人民共和国国家标准出版物上数字用法的规定》，1995
[2]《正确填写票据和结算凭证的基本规定》，2005

第 5 章 三个水桶等分 8 升水的问题

有这样一道智力题目：有三个容积分别是 3 升、5 升和 8 升的水桶，其中容积为 8 升的水桶中装满了水，容积为 3 升和容积为 5 升的水桶是空的。三个水桶都没有体积刻度，现在需要将大水桶中的 8 升水等分成两份，每份都是 4 升水，附加条件是只能使用另外两个空水桶，不能借助其他辅助容器。

这是一个很经典的问题，但是并不难，大部分人都可以在一分钟内给出答案。不过，很多人可能没有注意到，这个问题的答案不止一个。先来看一个最常见的答案，也是目前已知最快的操作步骤，共需要 7 次倒水动作：

(1) 从 8 升水桶中倒 5 升水到 5 升水桶中
(2) 从 5 升水桶中倒 3 升水到 3 升水桶中
(3) 从 3 升水桶中倒 3 升水到 8 升水桶中
(4) 从 5 升水桶中倒 2 升水到 3 升水桶中
(5) 从 8 升水桶中倒 5 升水到 5 升水桶中
(6) 从 5 升水桶中倒 1 升水到 3 升水桶中
(7) 从 3 升水桶中倒 3 升水到 8 升水桶中

最后的结果是 5 升水桶和 8 升水桶中各有 4 升水。再来看一个稍微复杂一点的答案，这个答案需要 8 次倒水动作：

(1) 从 8 升水桶中倒 3 升水到 3 升水桶中
(2) 从 3 升水桶中倒 3 升水到 5 升水桶中
(3) 从 8 升水桶中倒 3 升水到 3 升水桶中
(4) 从 3 升水桶中倒 2 升水到 5 升水桶中
(5) 从 5 升水桶中倒 5 升水到 8 升水桶中
(6) 从 3 升水桶中倒 1 升水到 5 升水桶中
(7) 从 8 升水桶中倒 3 升水到 3 升水桶中
(8) 从 3 升水桶中倒 3 升水到 5 升水桶中

到底有多少种答案？水从水桶之间倒来倒去，情况太多了，我这平凡的地球脑袋搞不定这个问题，但是计算机可以。设计一个算法，让计算机帮我们把所有的答案都找出来，这就是本章的内容。在写本书时我已经知道答案了，我没想到会有这么多种倒水方法。

5.1 问题与求解思路

如果用人的思维方式，那么解决这个问题的关键是怎么通过倒水凑出确定的 1 升水或能容纳 1 升水的空间，三只水桶的容积分别是 3、5 和 8，用这三个数做加减运算，可以得到很多组答案，例如：

$$3 - (5 - 3)=1$$

这个策略对应了上面提到的第一种解决方法，而另一组运算：

$$(3 + 3) - 5=1$$

则对应了上面提到的第二种解决方法。

但是计算机并不能理解这个“1”的重要性，很难按照人类的思维方式按部就班地推导答案，因此用计算机解决这个问题，通常会选择使用“穷举法”。为什么使用“穷举法”呢？因为这不是一个典型意义上的求解最优解的问题，虽然可能暗含了求解倒水次数最少的方法的要求，但就本质而言，常用的求解最优解问题的高效方法都不适用于此问题。如果能够穷举解空间的全部合法解，然后通过比较找到最优解也是一种求解最优解的方法。不过就本题题意而言，并不关心什么方法最快，能求出全部等分水的方法可能更符合题意。

使用“穷举法”，首先要定义问题的解，并分析解空间的范围和拓扑结构，然后根据解空间的范围和拓扑结构设计遍历搜索算法。如果我们把某一时刻三个水桶中存水的情况称为一个状态，则问题的初始状态是 8 升的水桶装满水，3 升和 5 升的水桶为空。最终要求的解的状态就是 3 升的水桶为空，5 升水桶和 8 升水桶各 4 升水。针对此问题的“穷举法”的实质就是从初始状态开始，根据某种状态变化的规则搜索全部可能的状态，每当找到一个从初始状态到最终状态的变化路径，就可以理解为找到了一个解，这条从初始状态到最终状态的路径就是倒水问题的一种答案。

状态都是静止的，从初始状态到最终状态的变化需要一种“推动力”，接下来需要找到一种“推动力”推动状态发生变化。这个“推动力”就是隐含在问题描述中的“倒水动作”，每个动作实施的结果就是从一个水桶倒水到另一个水桶，水桶中水的状态就发生变化了，于是状态也就变化了。如果能找到一种方式，持续地促使倒水动作发生，使得状态能不停地随动作变化，那就等于找到了本问题的解空间搜索方法。

5.2 建立数学模型

根据上一节的分析，求解这个问题的算法本质上就是对状态的穷举搜索。这样状态变化搜索的结果通常是得到一棵状态搜索树，根节点是初始状态，叶子节点可能是最终状态，也可能是某个无法转换到最终状态的中间状态。状态树有多少个最终状态的叶子节点，就有多少种答案。由此可知，解决本问题的算法关键是建立状态和动作的数学模型，并找到一种持续驱动动作产生的搜索方法。

本问题并不复杂，因此建立数学模型的工作就“退化”成建立描述问题的数据结构。前面定义的状态都是静止状态，完整的状态模型不仅要能够描述静止状态，还要能够描述并记录状态转换动作，尤其是对状态转换的描述，因为这会影响到状态树搜索算法的设计。先来看看状态模型以及状态树的设计。

5.2.1 状态的数学模型与状态树

所谓的静止状态，就是某一时刻三个水桶中存留水的体积。我们采用长度为 3 的一维向量描述这个状态，这组向量的三个值分别是容积为 8 升的桶中的水量、容积为 5 升的桶中的水量和容积为 3 升的桶中的水量。因此算法的初始状态就可以描述为[8, 0, 0]，则终止状态为[4, 4, 0]。

倒水动作与静止状态的结合就产生了状态变化，持续的状态变化就产生了一棵状态树，这个状态树上的所有状态就构成了穷举算法的解空间。以初始状态[8, 0, 0]为例，如果与“倒 5 升水到 5 升水桶”动作相结合，就得到了一个新状态[3, 5, 0]，同样，如果与“倒 3 升水到 3 升水桶”动作相结合，就得到了另一个新状态[5, 0, 3]。以此类推，可以得到如图 5-1 所示的状态树。

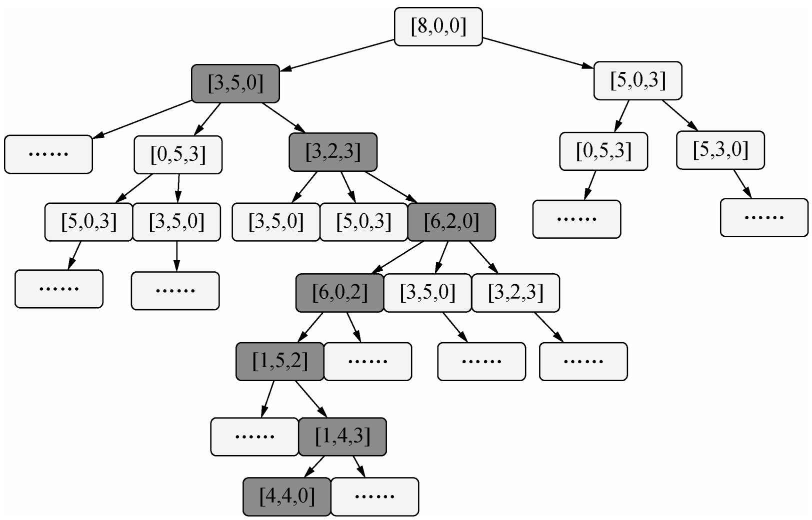


图 5-1 状态树结构示意图

[8, 0, 0]是状态树的根，图 5-1 只画出了这棵状态树的一部分，图中深颜色背景标识出的几个状态是状态树的一个分支，也是一个正确的解的状态转换路径。根据题目的意图，最终的结果是要输出这条转换路径的倒水过程，实际上就是与状态转换路径相对应的动作路径或动作列表。当定义了动作的数学模型后，就可以根据状态图中状态转换路径反推出对应的动作列表。依次输出这个动作列表就可以得到一个倒水过程的答案。

5.2.2 倒水动作的数学模型

两个静态状态是通过倒水动作建立关联的，这里说的倒水动作必须是合法的倒水动作。因为水桶是没有体积刻度的，因此倒水动作也就不能是任意的倒水行为。一个合法的倒水动作的前提条件是倒出水的桶中有水且倒入水的桶中还有空间。分析一下，实际上就两种情况，一种是倒入水的桶中的空间足够大，则倒出水的桶中的水全部加到倒入水的桶中，此时倒出水的桶成为空桶；另一种情况就是倒入水的桶中的空间不够大，只能倒一部分水，此时倒出水的桶中还剩有水。

一个合法的倒水动作包含三个要素：倒出水的桶、倒入水的桶和倒水体积。我们用一个三元组来描述倒水动作：{from, to, water}，from 是指从哪个桶中倒水，to 是指将水倒向哪个桶，water 是此次倒水动作所倒的水量。倒水动作的数据结构定义如下：

```
typedef struct tagACTION
{
    int from;
    int to;
    int water;
}ACTION;
```

某一时刻三个水桶中的存水状态，经过某个倒水动作后演变到一个新的存水状态，这是对状态转换的文字描述，对算法来讲，倒水状态描述就是“静止状态” + “倒水动作”。将静止状态和倒水动作组合在一起的原因是为了结果输出，因为此问题最终的答案是要求提供如何倒水的过程。包含动作的倒水状态定义如下。

```
struct BucketState
{
    …
    int bucket_s[BUCKETS_COUNT];
    ACTION curAction;
};
```

本模型的特例就是第一个状态如何得到，也就是[8, 0, 0]这个状态对应的倒水动作如何描述？我们用−1 表示未知的水桶编号（上帝水桶），因此第一个状态对应的倒水动作就是{−1, 1, 8}。应用本模型对前面提到的第一种解决方法进行状态转换描述，整个过程如图 5-2 所示。

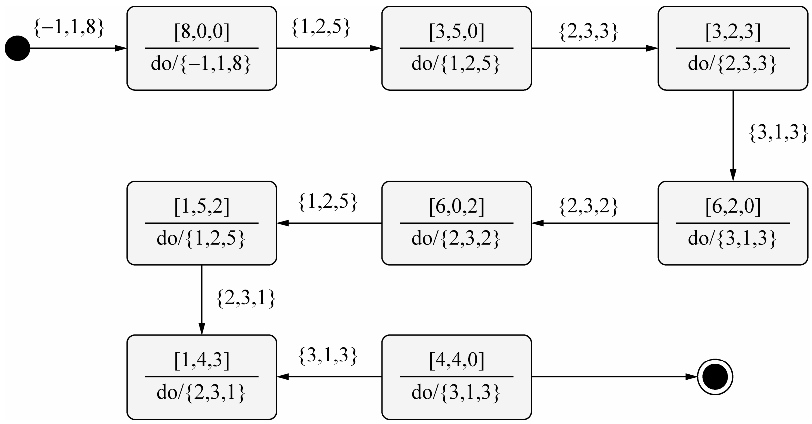


图 5-2　组合状态转换过程示意图

5.3　搜索算法

确定了状态模型后，就需要解决算法面临的第二个问题：状态树的搜索算法。一个静止状态结合不同的倒水动作会迁移到不同的状态，所有状态转换所展示的就是一棵如图 5-1 所示的状态树。对于本问题来说，这个状态树最初只有一个根节点，整棵树的展开是随着搜索算法逐步展开的。对于树状结构的搜索，可以采用深度优先搜索（DFS）算法，也可以采用广度优先搜索（BFS）算法。两种方法各有优缺点，广度优先搜索的优点是不会因为状态重复出现而导致搜索时出现状态环路，缺点是需要比较多的存储空间记录中间状态。深度优先搜索的优点是在同一时间只需要存储从根节点到当前搜索状态节点这一条路径上的状态节点，需要的存储空间比较小，缺点是要对搜索过程中因出现重复状态导致的状态环路做特殊处理，避免状态搜索时出现死循环的情况。

状态树的搜索就是对整个状态树进行遍历，其中暗含了状态的生成，因为状态树一开始并不完整，只有一个初始状态的根节点，当搜索（也就是遍历）操作完成时，状态树才完整。前面已经提到，树的遍历可以采用广度优先遍历算法，也可以采用深度优先遍历算法。就本题而言，要求解所有可能的等分水的方法，暗含了要记录从初始状态到最终状态，所以更适合使用深度优先遍历算法。

5.3.1　状态树的遍历

状态树的遍历暗含了状态生成的过程，就是促使状态树上的一个状态向下一个状态转换的驱动过程，这是一个很重要的部分，如果不能正确地驱动状态变化，就不能实现状态树的遍历（搜索）。建立状态模型一节中提到的动作模型，就是驱动状态变化的关键因子。对一个状态来说，它能转换到哪些新状态，取决于它能应用哪些倒水动作，一个倒水动作能够在原状态的基础上“生成”一个新状态，不同的倒水动作可以“生成”不同的新状态。由此可知，状态树遍历的关键是找到三个水桶之间所有合法的倒水动作，用这些倒水动作分别“生成”各自相应的新状态。

遍历三个水桶所有可能的倒水动作，就是对三个水桶任取两个进行全排列，这种排列的结果可以得到 6 种水桶的排列关系，这也就意味着有 6 种可能的倒水动作。将这 6 种倒水动作依次应用到当前状态，就可以“生成”6 种新状态，从而驱动状态发生变化。但是，受当前水桶的状态的影响，并不是 6 种排列关系都能组合出合法的倒水动作。例如，我们给三个水桶编号 1、2、3，取 1 号水桶和 3 号水桶得到一个排列关系（1,3），意味着从 1 号水桶向 3 号水桶倒水，但是如果 1 号水桶没有水，或者 3 号水桶已经满了，则无法进行从 1 号水桶向 3 号水桶倒水的动作。因此，在组合倒水动作时，需要结合当前三个水桶的存水状态判断是否是合法的倒水动作。BucketState::CanTakeDumpAction()函数负责做这个判断，其实现代码如下：

```
bool BucketState::CanTakeDumpAction(int from, int to)
{
    assert((from >= 0) && (from < BUCKETS_COUNT));
    assert((to >= 0) && (to < BUCKETS_COUNT));

/*不是同一个桶，且 from 桶中有水，且 to 桶中不满*/
    if( (from != to)
        && !IsBucketEmpty(from)
        && !IsBucketFull(to) )
    {
        return true;
    }

    return false;
}
```

from 是倒出水的水桶编号，to 是接收水的水桶编号。判断的依据有三个：第一，不能向自身倒水；第二，倒出水的桶不能为空桶；第三，接收水的桶必需有空间接收水，不能是满桶状态。

5.3.2 剪枝和重复状态判断

上一节提到，采用深度优先搜索状态树，会遇到重复状态导致的状态环路。比如，假设某一时刻从 1 号桶倒 3 升水到 3 号桶，下一个时刻又从 3 号桶倒 3 升水到 1 号桶，此时水桶的状态就又回到了之前的状态，这就形成一个状态环路。有时候状态环路可能复杂一点，几个状态之后才出现重复状态，图 5-1 展示的就是一种复杂一点的状态环路。在状态[3, 5, 0]→[3, 2, 3]→[6, 2, 0]→[3, 5, 0]的转换过程中，[3, 5, 0]状态再次出现形成状态环路。如果对这种情况不做处理，状态搜索就会在某个状态树分支陷入死循环，永远无法到达正确的结果状态。除此之外，如果对一个状态树分支上的某个状态经过搜索，其结果已经知道，则在另一个状态树分支上搜索时再遇到这个状态时，可以直接给出结果，或跳过搜索，以便提高搜索算法的效率。在这个过程中因重复出现被放弃或跳过的状态，可以理解为另一种形式的“剪枝”，可以使一次深度优先遍历很快收敛到初始状态。

考虑到上述两种情况，需要对当前深度遍历过程中经过的搜索路径上所有已经搜索过的状态做一个记录，形成一个当前已经处理过的状态表。每次因为动作组合生成新状态时，都检查一下

是否在这个记录中有状态相同的记录，如果存在状态相同的记录则跳过这个新状态，回溯到上一步继续处理下一个状态。如果新状态是状态表中没有的状态，则将新状态加入到状态表，然后从新状态开始继续深度优先遍历。

本问题还有一个要求，就是在搜索到一个最终状态时，输出搜索过程中记录的状态，以便还原整个过程的倒水动作。这也需要一个列表用于记录一次深度优先遍历过程中已经处理过的状态，算法设计时可以考虑将这两个表合二为一。如此一来，这个存放状态记录的列表不仅要支持从一端插入和删除状态，还要支持从头到尾遍历所有记录。从这两方面考虑，我们采用双端队列数据结构来维护这个记录列表。利用 C++的 STL 提供的便利，可以很简单地实现状态重复判断的算法，IsProcessedState()函数就是算法的实现代码。

```
bool IsProcessedState(std::deque<BucketState>& states, const BucketState& newState)
{
    std::deque<BucketState>::iterator it = states.end();

    it = find_if( states.begin(), states.end(),
         std::bind2nd(std::ptr_fun(IsSameBucketState), newState) );

    return (it != states.end());
}
```

find_if()算法需要一个仿函数，我不想再写一个函数对象，只好利用两个函数适配器重用了一个已经存在的普通函数 IsSameBucketState()。如果有 C++ 11 的编译器，可以利用 lamda 表达式改写这个算法。

5.4 算法实现

状态树的搜索是一个递归实现的过程：从初始状态开始，由第一个合法的倒水动作得到一个新的状态，记录这个状态，并从这个新状态开始递归搜索。在一个分支搜索完成后（无论是否得到结果），取消这个状态，然后从下一个合法的倒水动作再得到一个新状态，然后从这个状态开始继续搜索，直到遍历完所有合法的倒水动作。

这是一个递归算法，状态树搜索必须有一个终止条件，否则算法无法收敛。那么本问题的状态搜索的终止条件是什么？这要从两方面看，一方面是倒水动作的遍历，这是一个排列组合问题，排列完所有组合就是结束条件。另一方面是状态判断，如果出现了等分水的最终状态，则可以结束对状态树上当前分支的搜索。

SearchState()函数就是状态搜索算法的核心，这个函数首先检查当前状态列表的最后一个状态是否是结果需要的最终状态（[4, 4, 0]），如果是最终状态，就表示搜索到一个结果，通过调用 PrintResult()函数遍历状态列表，输出当前结果状态转换的整个过程（倒水动作序列）。如果当前状态不是最终状态，就通过一个两重循环遍历 6 种可能的倒水动作，将这些动作分别与当前状态结合形成新的状态，然后继续搜索新的状态。

```
void SearchState(std::deque<BucketState>& states)
{
    BucketState current = states.back(); /*每次都从当前状态开始*/
    if(current.IsFinalState())
    {
        PrintResult(states);
        return;
    }

    /*使用两重循环排列组合种倒水状态*/
    for(int j = 0; j < BUCKETS_COUNT; ++j)
    {
        for(int i = 0; i < BUCKETS_COUNT; ++i)
        {
            SearchStateOnAction(states, current, i, j);
        }
    }
}
```


搜索算法的递归关系是通过 SearchStateOnAction()函数实现的，首先调用 BucketState::CanTakeDumpAction()函数判断能否组合一个从 from 到 to 的倒水动作，然后调用 BucketState::DumpWater()函数实现倒水动作，并得到一个新状态。接着调用 IsProcessedState()函数判断这个状态是否是被处理过的状态，如果是没有被处理过的新状态，则将这个新状态加入到已搜索状态记录表，并调用 SearchState()函数继续搜索。

```
void SearchStateOnAction(std::deque<BucketState>& states, BucketState& current, int from, int to)
{
    if(current.CanTakeDumpAction(from, to))
    {
        BucketState next;
         /*从 from 到 to 倒水，如果成功，返回倒水后的状态*/
        bool bDump = current.DumpWater(from, to, next);
        if(bDump && !IsProcessedState(states, next))
        {
            states.push_back(next);
            SearchState(states);
            states.pop_back();
        }
    }
}
```

BucketState::DumpWater()也是一个很有意思的函数。前面介绍的 BucketState::CanTakeDumpAction()函数只是判断从 from 到 to 能否组合出倒水动作，而这个函数则是完成实际倒水动作的具体算法实现。首先计算 to 水桶的剩余容积，然后根据 from 水桶中的水量决定本次能倒多少水，如果 from 水桶中剩余水量比 to 水桶中的剩余容积小，则 from 水桶被倒空。真正的倒水动作其实就从 from 桶中减去倒水量，在 to 水桶加上对应的倒水量。如果倒水成功，最后调用 BucketState::SetAction()函数将倒水动作三元组与新状态绑定，得到一个动态的倒水动作状态，新状态通过 next 参数返回。

```
bool BucketState::DumpWater(int from, int to, BucketState& next)
```

```
{
    next.SetBuckets(bucket_s);
    int dump_water = bucket_capicity[to] - next.bucket_s[to];
    if(next.bucket_s[from] >= dump_water)
    {
        next.bucket_s[to] += dump_water;
        next.bucket_s[from] -= dump_water;
    }
    else
    {
        next.bucket_s[to] += next.bucket_s[from];
        dump_water = next.bucket_s[from];
        next.bucket_s[from] = 0;
    }
    if(dump_water > 0) /*是一个有效的倒水动作?*/
    {
        next.SetAction(dump_water, from, to);
        return true;
    }

    return false;
}
```

5.5　总结

本章开始给出了三个水桶等分 8 升水问题的两个答案，实际答案不止两个。我用图 5-1 所示的画状态图的方法手推答案，推算到第 6 种方法的时候就放弃了。需要搜索的状态很多，需要逐个判断状态的处理情况，还是让计算机做吧。用本章的算法穷举之后，一共找到了 16 种倒水的方法，最快的方法需要 7 个步骤，也就是本章给出的第一种方法。如果不用算法，你能给出几种倒水方法呢？试试看吧。

5.6　参考资料

[1]　Levitin A. 算法设计与分析基础. 潘彦译. 北京：清华大学出版社，2007

[2]　Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[3]　Knuth D E. *The Art of Computer Programming* (*Third Edition*)，Vol 2. Addison-Wesley, 1997

第 6 章 妖怪与和尚过河问题

这是一个从 www.plastelina.net 网站下载的 Flash 小游戏，如图 6-1 所示。有三个和尚和三个妖怪（也可翻译为传教士和食人妖）要利用唯一一条小船过河，这条小船一次只能载两个人，同时，无论是在河的两岸还是在船上，只要妖怪的数量大于和尚的数量，妖怪们就会将和尚吃掉。现在需要选择一种过河的安排，保证和尚和妖怪都能过河且和尚不能被妖怪吃掉。

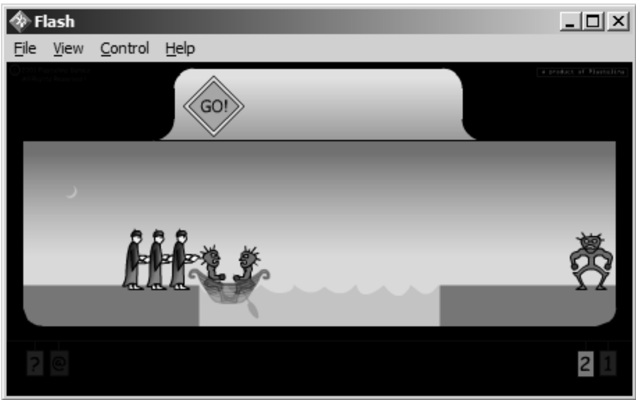


图 6-1　妖怪与和尚过河游戏

这其实是一个很简单的游戏，过河的策略就是无论何时都要保证在河的任意一侧和尚数量多于妖怪。先来看一种过河的方法。

(1) 两个妖怪先过河，一个妖怪返回；
(2) 再两个妖怪过河，一个妖怪返回；
(3) 两个和尚过河，一个妖怪和一个和尚返回；
(4) 两个和尚过河，一个妖怪返回；
(5) 两个妖怪过河，一个妖怪返回；
(6) 两个妖怪过河。

我现在知道，这个游戏的答案不止一个，到底有几个答案呢？写个算法来找找吧。

6.1 问题与求解思路

题目的初始条件是三个和尚和三个妖怪在河的一边，和它们在一起的还有一条小船。过河后的情况应该是三个和尚和三个妖怪安全地过到河的对岸，虽然没有明确提到船的状态，但是船也应该跟着到了对岸，否则岂不闹鬼了？我们看这个问题里的三个关键因素，就是和尚、妖怪和小船，当然，还有它们的位置。假如我们要让计算机理解这个问题，除了对这三个事物进行描述，还要定义它们的位置信息。如果把任意时刻妖怪、和尚和小船的位置信息合在一起看作一个“状态”，则要解决这个问题只需要找到一条从初始状态变换到终止状态的路径即可。这就有点类似于第 5 章介绍的用三个水桶等分 8 升水的问题，我们可以尝试使用第 5 章介绍的穷举方法，遍历所有由妖怪、和尚和小船的位置构成的状态空间，寻找一条或多条从初始状态到最终状态的转换路径。

从初始状态开始，通过构造特定的搜索算法，对状态空间中的所有状态进行穷举，就得到一棵以初始状态为根的状态树。如果状态树上某个叶子节点是题目要求的最终状态，则从根节点到此叶子节点之间的所有状态节点就是一个过河问题的解决过程。从初始状态开始，每选择一批妖怪或和尚过河，就会从原状态产生一个新的状态。如果以人类思维解决这个问题，每次都会选择最佳的妖怪与和尚组合过河，使得它们过河后生成的新状态更接近最终状态，不断重复上述过程，直到得到最终状态。在这个过程中，人的选择是推动状态转换的驱动力。用计算机解决妖怪与和尚过河问题的思路也是通过状态转换，找到一条从初始状态到结束状态的转换路径。计算机不会进行理性分析，不知道每次如何选择最佳的过河方式，但是计算机擅长快速计算且不知疲劳，既然不知道如何选择过河方式，那就干脆把所有的过河方式都尝试一遍，找出所有可能的结果，当然也就包括成功过河的结果。也就是说，用计算机求解这个问题，穷举各种动作尝试就是推动状态变化的驱动力。

6.2 建立数学模型

本章介绍的算法和第 5 章的算法类似，都是从一个根状态开始对状态空间进行搜索，其结果也是一棵状态搜索树。解决本问题的算法关键是建立状态和动作的数学模型，并找到一种持续驱动动作产生的搜索方法。本问题并不复杂，因此建立数学模型的工作就“退化”成建立描述问题的数据结构。本问题的状态模型不仅要能够描述静止状态，还要能够描述并记录状态转换动作，尤其是对状态转换的描述，因为这会影响到状态树搜索算法的设计。除此之外，当搜索算法找到一个最终状态时，需要输出从开始状态到最终状态的动作序列，这也需要状态模型能够和动作模型结合在一起。下面一起来看看本问题的状态模型以及状态树的设计。

6.2.1 状态的数学模型与状态树

观察一下本问题的状态，看起来好像是 3 个和尚、3 个妖怪加上一只船一共 7 个属性，但是仔细研究就会发现，3 个和尚之间和 3 个妖怪之间没有差异，也没有顺序关系，因此在考虑数学模型的时候不需要赋予它们太多的属性，只要用数量表示它们就可以了。对于和尚和妖怪的状态，分别用两个值表示它们在河两岸的数量，这样只需 4 个属性就可以表示，分别是河左岸和尚数量、河左岸妖怪数量、河对岸和尚数量和河对岸妖怪数量。每当有妖怪或和尚随船的移动发生变化时，只需要修改和尚和妖怪在河两岸的数量即可完成状态的转换。除了和尚和妖怪的数量，还有一个关键因素也会影响到状态的变化，那就是小船的位置。小船的位置是个非常重要的状态属性，不仅决定了状态的差异，还会影响后序动作的选择。

最后的状态模型中，和尚与妖怪的状态就是数值，船有两个枚举状态，在河左岸（LOCAL）和在河对岸（REMOTE）。我们用一个五元组来表示某个时刻的过河状态：[本地和尚数，本地妖怪数， 对岸和尚数，对岸妖怪数，船的位置]。用五元组表示的初始状态就是[3, 3, 0, 0, LOCAL]，问题解决的过河状态是[0, 0, 3, 3, REMOTE]。和尚、妖怪和小船的状态模型定义的数据结构如下所示：

```
struct ItemState
{
    ...
    int local_monster;
    int local_monk;
    int remote_monster;
    int remote_monk;
    BOAT_LOCATION boat; /*LOCAL or REMOTE*/
    ...
};
```

状态模型确定以后，整个状态空间的树形模型也就确定了，读者可以参考第 5 章的图 5-1 理解一下本问题的状态树。接下来就要确定和尚与妖怪过河的动作模型，过河动作是驱动状态变化的关键。

6.2.2 过河动作的数学模型

河两岸的和尚与妖怪的数量发生变化的直接原因是小船的位置关系发生变化，因为船上至少要有一个和尚或妖怪，所以只要船的位置发生变化，必然会引起状态的变化。过河动作是促使船的位置发生变化的原因，也是连接两个状态的转换关系。这个转换关系包含两部分内容，一部分是船的位置变化，另一部分是船上的妖怪或和尚的数量，这个数量会引起两岸的和尚和妖怪的数量发生变化。

过河动作的数学模型需要明确定义两个内容，即动作引起船的位置变化情况和此动作移动的和尚或妖怪的数量。过河动作的具体数据结构定义如下：

```
typedef struct tagActionEffection
```

```
{
    ACTION_NAME act;
    BOAT_LOCATION boat_to; //船移动的方向
    int move_monster; //此次移动的妖怪数量
    int move_monk; //此次移动的和尚数量
}ACTION_EFFECTION;
```

ACTION_NAME 是一个比较有意思的属性，其实是对动作的一个命名。“三个水桶等分 8 升水问题”中的动作是通过排列组合三个水桶的关系产生的，但是过河问题没有这个条件，这也是同一类问题处理细节上的差异。虽然不能通过排列组合产生动作，但是通过对问题的观察，我们发现过河问题的所有过河动作其实是一个有限的动作集合。看一下 ACTION_EFFECTION 的定义，根据题目的要求，无论船是从左岸到对岸，还是从对岸返回到左岸，船上装载的妖怪和和尚的情况只能是以下五种：一个妖怪、一个和尚、两个妖怪、两个和尚以及一个妖怪加一个和尚。结合船移动的方向，一共只有 10 种过河动作可供选择，分别是：

- 一个妖怪过河
- 两个妖怪过河
- 一个和尚过河
- 两个和尚过河
- 一个妖怪和一个和尚过河
- 一个妖怪返回
- 两个妖怪返回
- 一个和尚返回
- 两个和尚返回
- 一个妖怪和一个和尚返回

于是，ACTION_NAME 的定义如下：

```
typedef enum tagActionName
{
    ONE_MONSTER_GO = 0,
    TWO_MONSTER_GO,
    ONE_MONK_GO,
    TWO_MONK_GO,
    ONE_MONSTER_ONE_MONK_GO,
    ONE_MONSTER_BACK,
    TWO_MONSTER_BACK,
    ONE_MONK_BACK,
    TWO_MONK_BACK,
    ONE_MONSTER_ONE_MONK_BACK,
    INVALID_ACTION_NAME,
}ACTION_NAME;
```

请注意，如果 ACTION_NAME 不同，其对应的 boat_to、move_monster 和 move_monk 三个属性也不相同。这个问题有 10 种不同的动作，如果对这 10 种动作不能用一个抽象的记录进行一致性处理，那么我们的算法代码就不可避免地出现长长的 if…else 语句或 switch…case 语句。代码中长的

if…else 或 switch…case 语句正是各种问题的起源，我们要尽量避免出现这种情况。怎么做一致性处理？这是算法设计中常用的技巧之一，总结起来就是两点：首先对要处理的数据进行归纳处理，确定共性的部分和差异的部分；然后对差异部分进行量化处理，将逻辑的差异转化成计算机能一致性处理的差异，比如数字的大小变化、字符串的长短变化，等等。在本例中，动作名称和小船的位置是共性的部分，计算机已经不用区分动作的实际类型就可以进行一致处理。和尚和妖怪的移动方法随动作类型不同而变化，无法统一处理，但是可以转化成数字的加减法处理。举个例子，一个和尚和一个妖怪过河的动作，实际效果就是河左岸的和尚数量和妖怪数量各减一，河对岸的和尚数量和妖怪数量各加一。整理起来，所有的动作可归纳为以下动作列表：

```
ACTION_EFFECTION actEffect[] =
{
    { ONE_MONSTER_GO ,           REMOTE, -1,  0 },
    { TWO_MONSTER_GO ,           REMOTE, -2,  0 },
    { ONE_MONK_GO ,              REMOTE,  0, -1 },
    { TWO_MONK_GO ,              REMOTE,  0, -2 },
    { ONE_MONSTER_ONE_MONK_GO ,  REMOTE, -1, -1 },
    { ONE_MONSTER_BACK ,         LOCAL ,  1,  0 },
    { TWO_MONSTER_BACK ,         LOCAL ,  2,  0 },
    { ONE_MONK_BACK ,            LOCAL ,  0,  1 },
    { TWO_MONK_BACK ,            LOCAL ,  0,  2 },
    { ONE_MONSTER_ONE_MONK_BACK , LOCAL ,  1,  1 }
};
```

列表中的 move_monster 属性和 move_monk 属性如果是负数，则表示是从本地移动到河对岸。这个动作列表是我们进行状态转换一致性处理的基础，直接使用这张表就不需要对每种动作都进行特殊处理，可以避免使用长长的 if…else 或 switch…case 语句。

6.3 搜索算法

本章介绍的算法仍然采用深度优先遍历算法，每次遍历只暂时保存当前搜索的分支的所有状态，之前搜索过的分支上的状态是不保存的，只在必要的时候输出结果。因此，算法不需要完整的树状数据结构保存整个状态树（也没有必要这么做），只需要一个队列能暂时存储当前搜索分支上的所有状态即可。这个队列初始时只有一个初始状态，随着搜索的进行逐步增加，当搜索算法完成后，队列中应该仍然只有一个初始状态。状态树的搜索过程就是状态树的生成过程，因此状态树一开始并不完整，只有一个初始状态的根节点，当搜索（也就是遍历）操作完成时，状态树才完整。

一个静止状态结合不同的过河动作会迁移到不同的状态。上一节已经分析过了，每个状态所能采用的过河动作只能是 ActionName 标识的 10 种动作中的一种（当然并不是每种动作都适用于此状态），有了这个动作范围，搜索状态树的穷举算法就非常简单了，只需将当前状态分别与这 10 种动作进行组合，就可以得到状态树上这个状态所有可能的新状态，对新状态继续应用各种过河动作，再得到新状态，直到出现最终状态，得到一个过河过程。图 6-2 就是一个过河结果的状态转换过程。

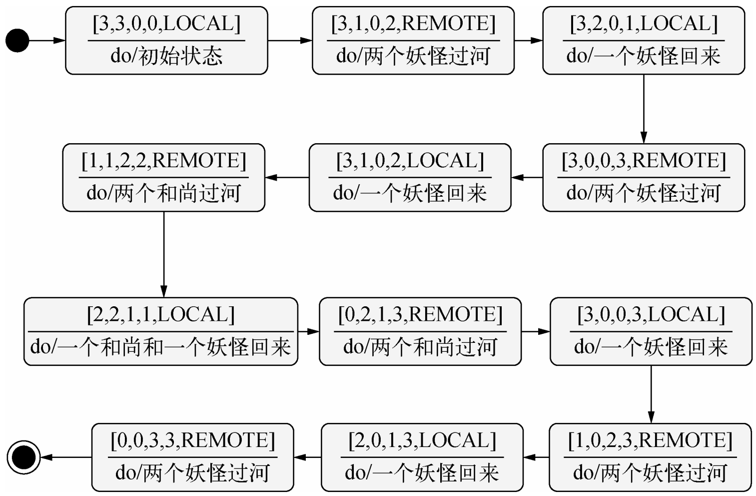


图 6-2　一个过河结果的状态转换过程

6.3.1　状态树的遍历

状态树的遍历暗含了一个状态生成的过程，就是促使状态树上的一个状态向下一个状态转换的驱动过程，这是一个很重要的部分，如果不能正确地驱动状态变化，就不能实现状态树的遍历（搜索）。建立状态模型一节中提到的动作模型，就是驱动状态变化的关键因子。算法的动作模型一共定义了 10 种动作，每种动作结合当前状态就可以产生一个新的状态，就可以推动状态产生变化。当然，并不是所有的动作都能适用于当前状态，比如，假设当前状态是只有两个妖怪在河左岸，则“一个和尚过河”“两个和尚过河”和“一个和尚和一个妖怪过河”这三种动作就不适用于当前状态。

状态树遍历的关键就是处理过河动作列表 actEffect，依次处理一遍这个列表中的每个动作就实现了状态树的搜索，因为使用了表结构，代码变得非常简单：

```
/*尝试用 10 种动作分别与当前状态组合*/
for(int i = 0; i < sizeof(actEffect) / sizeof(actEffect[0]); i++)
{
    ProcessStateOnNewAction(states, current, actEffect[i]);
}
```

6.3.2　剪枝和重复状态判断

前面已经提到过，并不是所有的动作都适用于当前状态，那么，如何判断一个动作是否适用于当前状态？首先，当前状态中船的位置很关键，如果船的位置在河对岸，那么所有的过河动作就都不适用。其次是移动的妖怪或和尚的数量是否与当前状态相适应，比如 6.3.1 节给出的例子，如果河左岸没有和尚，那么所有需要移动和尚的动作就都不适用。根据以上分析，我们可以给出

判断动作合法性的算法：

```
bool ItemState::CanTakeAction(ACTION_EFFECTION& ae) const
{
    if(boat == ae.boat_to)
        return false;
    if((local_monster + ae.move_monster) < 0
        || (local_monster + ae.move_monster) > monster_count)
        return false;
    if((local_monk + ae.move_monk) < 0
        || (local_monk + ae.move_monk) > monk_count)
        return false;

    return true;
}
```

应用这个判断，可以省去很多不必要的状态变化，避免出现一些不符合题目要求的错误状态，比如河左岸有–1 个和尚，河对岸有 4 个和尚这种情况。

本算法采用深度优先原则搜索状态树，就会遇到和“三个水桶等分 8 升水问题”算法一样的问题，那就是重复出现的状态会导致状态环路。比如某一时刻采用的动作是“一个和尚和一个妖怪过河”，到了河对岸形成新的状态，如果新状态采用的动作是“一个和尚和一个妖怪返回”，则最后的状态就变成了过河之前的状态，这两个状态加上这两个动作就会形成状态环路，搜索路径上存在状态环路的后果就是搜索算法可能会陷入死循环。除此之外，如果对一个状态树分支上的某个状态经过搜索，其结果已经知道，则在另一个状态树分支上搜索时再遇到这个状态时，可以直接给出结果，或跳过搜索，以便提高搜索算法的效率。在这个过程中因重复出现被放弃或跳过的状态，可以理解为另一种形式的“剪枝”，可以使一次深度优先遍历很快收敛到初始状态。

本算法依然采用双端队列来组织搜索过程中的已处理状态，相关的判断算法和“三个水桶等分 8 升水问题”的算法完全一样。

6.4 算法实现

算法的核心依然是递归搜索，从初始状态开始调用 SearchState()函数。函数每次从状态队列尾部取出当前要处理的状态，首先判断是否是最终的过河状态，如果是则输出一组过河方案，如果不是，则尝试用动作列表中的动作与当前状态结合，看看是否能生成合法的新状态。

```
void SearchState(std::deque<ItemState>& states)
{
    ItemState current = states.back(); /*每次都从当前状态开始*/
    if(current.IsFinalState())
    {
        PrintResult(states);
        return;
    }

    /*尝试用 10 种动作分别与当前状态组合*/
```

```
    for(int i = 0; i < sizeof(actEffect) / sizeof(actEffect[0]); i++)
    {
        SearchStateOnNewAction(states, current, actEffect[i]);
    }
}
```

搜索的递归关系是通过 SearchStateOnNewAction()函数体现的，这个函数首先判断当前状态和制定的过河动作是否能生成一个新状态，如果能得到一个合法的新状态，则继续处理这个新状态。

```
void SearchStateOnNewAction(std::deque<ItemState>& states,
    ItemState& current, ACTION_EFFECTION& ae)
{
    ItemState next;
    if(MakeActionNewState(current, ae, next))
    {
        if(next.IsValidState() && !IsProcessedState(states, next))
        {
            states.push_back(next);
            SearchState(states);
            states.pop_back();
        }
    }
}
```

MakeActionNewState()函数是一个很有意思的函数，它就是这个算法设计的通过过河动作属性列表对所有动作进行一致性处理的体现，通过对属性的直接加或减计算，避免了长 if…else 语句或 switch…case 代码。

```
bool MakeActionNewState(const ItemState& curState, ACTION_EFFECTION& ae, ItemState& newState)
{
    if(curState.CanTakeAction(ae))
    {
        newState = curState;
        newState.local_monster  += ae.move_monster;
        newState.local_monk     += ae.move_monk;
        newState.remote_monster -= ae.move_monster;
        newState.remote_monk    -= ae.move_monk;
        newState.boat     = ae.boat_to;
        newState.curAct   = ae.act;

        return true;
    }

    return false;
}
```

6.5 总结

最后，这个算法告诉我们一共有四种过河方案。大多数人都能很容易地给出本章开始时给出的方案，这个应该是最容易想到的。事实上，我也见过有人给出其他方案，所以我知道这个问题可能不止一种解决方案。现在，我们知道了，这个问题一共有四种解决方案，并且只有四种。

6.6 参考资料

[1] Levitin A. 算法设计与分析基础. 潘彦译. 北京：清华大学出版社，2007

[2] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[3] Knuth D E. *The Art of Computer Programming*(*Third Edition*) Vol 2. Addison-Wesley, 1997

第 7 章 稳定匹配与舞伴问题

每年凤凰花开、蝉鸣绿叶的季节，都是毕业的季节，也是同学们找工作的季节。很显然，学生和雇主之间从来都是双向选择的关系，然而学霸们往往先人一步，早早地就抓了一把 offer。无奈，即便是学霸也分身无术，最终只能选择一个 offer。毫无疑问，学霸们会根据自己的偏好对 offer 排队，选其中最好的一个。有时候我会想，其他也给了学霸 offer 的公司岂不是少了一个名额？显然我是多虑了，其实这些雇主公司也有一个偏好列表作为备用，如果空出了名额，他们会从这个备用的偏好队列中再选一个。但这总归不是一个最高效的资源配置方式，大量的撤销和重新选择会浪费很多社会资源。有没有一种方法，在双向选择公开透明的基础上，按照资源配置的最优原则给学生和雇主配对，直接得到一个学生和雇主之间的完备匹配或稳定匹配？

幸运的是，确实有人在研究稳定匹配问题（stable matching problem）。戴维·盖尔（David Gale）和劳埃德·沙普利（Lloyd Shapley）就是两位这样的专家，他们从 20 世纪 60 年代就开始研究这个问题。他们最早研究的问题是稳定婚姻问题（stable marriage problem），其实这适用于所有带偏爱或优先选择的双向选择问题。本章我们就以稳定婚姻问题为例，介绍一下盖尔和沙普利研究的稳定匹配算法（Gale-Shapley 算法）的原理，并给出一个解决舞伴匹配问题的 Gale-Shapley 算法实现。

7.1 稳定匹配问题

1962 年，盖尔和沙普利发表了一篇名为“大学招生与婚姻的稳定性”[1]的论文，首次提出了稳定婚姻问题，该问题后来成为研究稳定匹配的典型例子。在介绍稳定匹配问题之前，我们先来了解几个概念。

7.1.1 什么是稳定匹配

假设 n 个未婚男人的集合 $M=\{m_1,m_2,\cdots,m_n\}$ 和 n 个未婚女人的集合 $W=\{w_1,w_2,\cdots,w_n\}$，令 $M\times W$ 为所有可能的形如(m_i,w_i)的有序对的集合，其中 $m_i\in M$，$w_i\in W$。根据上述定义，我们给出匹配

的概念，匹配 S 是来自 $M \times W$ 的有序对的集合，并且具有以下性质：每个 M 的成员和每个 W 的成员至多出现在 S 的一个有序对中。接下来是完美匹配的概念，完美匹配 S'是一个具有以下性质的匹配：M 的每个成员和 W 的每个成员恰好出现在 S'的一个对里。S 和 S'这两个定义的差别就是“至多”和“恰好”两个词，对很多人来说，区分这两个概念就像区分落基山大角羊和沙漠大角羊一样困难。我来解释一下，可以将 S 理解为 M 和 W 的成员配对结婚，但是 M 和 W 中不一定所有成员都能配对成功，还有剩余的男人和女人是单身。而完美匹配 S'则是 S 的一种特殊情况，即 S'是所有人都配对成功，不存在落单的男人和女人。

很显然，盖尔和沙普利研究的稳定婚姻问题是在一夫一妻制度下男人和女人的配对关系，每个男人最终都要和一个女人结婚。现在在完美匹配的背景下引入优先或偏好的概念，每个男人都按照个人喜好对所有女人排名，如果某个男人 m 给女人 w 的排名高于给 w'的排名，就可以理解为 m 喜欢 w 胜过 w'。反过来也一样，每个女人也按照自己的喜好对所有的男人排名。以上排名必须区分先后顺序，不能有排名并列的情况出现。那么什么是稳定匹配呢？**稳定匹配**就是在引入优先排名的情况下，一个完美匹配 S 如果不存在不稳定因素，则称这个完美匹配是稳定匹配。什么是不稳定因素呢？假设在完美匹配 S 中存在两个配对（m, w）和（m', w'），但是从优先排名上看，m 更喜欢 w'而不喜欢 w，同时 w'也更喜欢 m 而不喜欢 m'，如图 7-1 所示。在这种情况下，我们称这个完美匹配 S 是不稳定的，像（m, w'）这样有“私奔”倾向的不稳定对（unstable pair）就是 S 的一个不稳定因素。

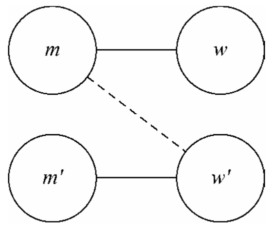


图 7-1 不稳定因素示意图

稳定匹配满足两个条件：首先，它是一个完美匹配；其次，它不含有任何不稳定因素。在给定的众多复杂关系中，如何求得一个稳定匹配？盖尔和沙普利在 1962 年提出的 Gale-Shapley 算法就是一种著名的稳定匹配算法，接下来我们就来简单介绍一下 Gale-Shapley 算法的原理。

7.1.2 Gale-Shapley 算法原理

盖尔和沙普利的策略是一种寻找稳定婚姻的策略，不管男女之间有何种偏好，这种策略总可以得到一个稳定的婚姻匹配。先来看一下 Gale-Shapley 算法实现的伪代码：

```
初始化所有的m∈ M, w∈ W, 所有的m和w都是自由状态;
while (存在男人是自由的, 并且他还没有对每个女人都求过婚)
{
    选择一个这样的男人m;
    w = m的优先选择表中还没有求过婚的排名最高的女人;
    if (w 是自由状态)
    {
        将(m, w)的状态设置为约会状态;
    }
    else /*w 已经和其他男人约会了*/
    {
        m' = w当前约会的男人;
        if (w 更喜爱m' 而不是m)
```

```
            {
                m 保持单身状态（w 不更换约会对象）;
            }
            else /*w 更喜爱 m 而不是 m'*/
            {
                将(m, w)的状态设置为约会状态;
                将 m' 设置为自由状态;
            }
        }
    }
    输出已经匹配的集合 S;
```

看起来总是男人主动选择，女人被动接受，事实上这个算法并没有做这个假设。基于男女平等的原则，也可以是女人主动选择，男人被动接受，这就是这个算法常被提到的两个策略，即“男士优先”还是“女士优先”。

从 Gale-Shapley 算法的策略来看，男人们一轮一轮地选择自己中意的女人，女人则可以选择接受追求者，或拒绝追求者。只要女人是单身的自由状态，男人的追求就不会被拒绝，但这并不表示男人总是能选到自己最中意的女人，因为女人是可以毁约的。男人被拒绝的情况有两种，一种情况是男人追求的女人已经有约会对象，并且女人喜欢自己的约会对象胜过当前追求她的男人；另一种情况是女人面对另一个男人的追求时，如果她喜欢这个追求她的男人胜过自己当前的约会对象，女人会利用毁约的权利拒绝当前约会对象。男人每被拒绝一次，就只能从自己的优先选择表中选择下一个女人。男人不能重复尝试约会那些已经拒绝过他的女人，因此这种选择总是无奈地向越来越不中意的方向发展。每一轮选择之后都会有一些男人或女人脱离单身的自由状态，当某一轮过后没有任何一个男人或女人是单身状态时，这个算法就结束了。在 Gale-Shapley 算法中，n 个男人共需要进行 n 轮选择，每一个男人需要向 n 个中意对象求婚，因此，算法最多需要 $n*n$ 轮循环就可以结束。

这个算法的流程非常简单，但是是否有效呢？也就是说 Gale-Shapley 算法结束后得到的一个匹配一定是稳定匹配吗？还记得上一节介绍的稳定匹配的两个条件吗？稳定匹配首先是完美匹配，其次是要求没有不稳定因素。下面我们就从这两方面分别证明一下这个算法的结果是否是稳定匹配。

首先，我们要证明 Gale-Shapley 算法结束得到的是一个完美匹配。直接证明这个问题比较困难，所以我们采用反证法。假设算法结束后有一个男人 m 还是单身，因为规则是一个男人只能和一个女人约会，这就意味着必定有一个女人 w 也是单身。根据算法规则，女人只要是单身，一定会接受男人的求婚，现在 w 是单身，说明 w 没有收到任何求婚请求。这时就出现矛盾了，因为根据算法流程，m 肯定是向包括 w 在内的所有女人都求过婚的，所以假设应该是不成立的，也就是说，能够证明 Gale-Shapley 算法得到的是一个完美匹配。

接下来证明 Gale-Shapley 算法的结果没有任何不稳定因素，仍然采用反证法。假设匹配结果中存在不稳定因素，也就是说，存在 m 和 w，他们各自都已经有了伴侣，但是 m 喜欢 w 胜过喜欢自己现在的伴侣，同样，w 也喜欢 m 胜过喜欢自己现在的伴侣。但是根据算法规则，m 肯

定是向 w 求过婚的，如果 w 更喜欢 m，w 应该选择 m 而不是当前的伴侣，因此这个假设也是不成立的。

由以上证明可知，Gale-Shapley 算法的结果是一个稳定匹配，也就证明了 Gale-Shapley 算法的正确性。

7.2　Gale-Shapley 算法的应用实例

本节利用舞伴问题介绍一个 Gale-Shapley 算法的应用实例。舞伴问题是这样的：有 n 个男孩与 n 个女孩参加舞会，每个男孩和女孩均交给主持一个名单，写上他（她）中意的舞伴名字。无论男孩还是女孩，提交给主持人的名单都是按照偏爱程度排序的，排在前面的都是他们最中意的舞伴。试问主持人在收到名单后，是否可以将他们分成 n 对，使每个人都能和他们中意的舞伴结对跳舞？为了避免舞会上出现不和谐的情况，要求这些舞伴的关系是稳定的。

假如有两对分好的舞伴：（男孩 A，女孩 B）和（男孩 B，女孩 A），但是男孩 A 更偏爱女孩 A，女孩 A 也更偏爱男孩 A，同样，女孩 B 更偏爱男孩 B，而男孩 B 也更偏爱女爱 B。在这种情况下，这两对舞伴就倾向于分开，然后重新组合，这就是不稳定因素。很显然，这个问题需要的是一个稳定匹配的结果，适合使用 Gale-Shapley 算法。

7.2.1　算法实现

首先定义舞伴的数据结构，根据题意，一个舞伴至少要包含两个属性，就是每个人的偏爱舞伴列表和他（她）们当前选择的舞伴。根据 Gale-Shapley 算法的规则，还需要有一个属性表示下一次要向哪个偏爱舞伴提出跳舞要求。当然，这个属性并不是男生和女生同时需要的，当使用“男士优先”策略时，男生需要这个属性，当使用“女士优先”策略时，女生需要这个属性。为了使程序输出更有趣味，需要为每个角色提供一个名字。综上所述，舞伴的数据结构定义如下：

```
typedef struct tagPartner
{
    char *name;   //名字
    int next;     //下一个邀请对象
    int current;  //当前舞伴，-1表示还没有舞伴
    int pCount;   //偏爱列表中舞伴个数
    int perfect[UNIT_COUNT]; //偏爱列表
}PARTNER;
```

UNIT_COUNT 是男孩或女孩的数量（稳定匹配问题总是假设男孩和女孩的数量相等），pCount 是偏爱列表中的舞伴个数。根据标准的“稳定婚姻问题”的要求，pCount 的值应该是和 UNIT_COUNT 一致的，但是某些情况下（比如一些算法比赛题目的特殊要求）也会要求伙伴们提供的偏爱列表可长可短，因此我们增加了这个属性。但是有一点需要注意，如果允许舞伴的 pCount 小于 UNIT_COUNT，则 7.1.2 节的证明就不适用了，需要设置相应的条件并使用更复杂的证明方法。关键是，最后不一定能得到稳定匹配的结果。这里给出的实现算法使用数组来存储参加舞会的男孩和

女孩列表，因此这个数据结构中的 next、current 和 perfect 列表中存放的都是数组索引，了解这一点有助于理解算法的实现代码。

Gale-Shapley 算法的实现非常简单，将 7.1.2 节给出算法伪代码翻译成编程语言即可。完整的算法代码如下：

```
bool Gale_Shapley(PARTNER *boys, PARTNER *girls, int count)
{
    int bid = FindFreePartner(boys, count);
    while(bid >= 0)
    {
        int gid = boys[bid].perfect[boys[bid].next];
        if(girls[gid].current == -1)
        {
            boys[bid].current = gid;
            girls[gid].current = bid;
        }
        else
        {
            int bpid = girls[gid].current;
            //女孩喜欢 bid 胜过其当前舞伴 bpid
            if(GetPerfectPosition(&girls[gid], bpid) > GetPerfectPosition(&girls[gid], bid))
            {
                boys[bpid].current = -1; //当前舞伴恢复自由身
                boys[bid].current = gid; //结交新舞伴
                girls[gid].current = bid;
            }
        }
        boys[bid].next++; //无论是否配对成功，对同一个女孩只邀请一次
        bid = FindFreePartner(boys, count);
    }

    return IsAllPartnerMatch(boys, count);
}
```

FindFreePartner()函数负责从男孩列表中找一个还没有舞伴、并且偏好列表中还有没有邀请过的女孩的男孩，返回男孩在列表（数组）中的索引。如果返回值等于–1，表示没有符合条件的男孩了，于是主循环停止，算法就结束了。GetPerfectPosition()函数用于判断女孩喜欢一个舞伴的程度，通过返回舞伴在自己的偏爱列表中的位置来判断，位置越靠前，也就是 GetPerfectPosition()函数的返回值越小，说明女孩越喜欢这个舞伴。GetPerfectPosition()函数的实现代码如下：

```
int GetPerfectPosition(PARTNER *partner, int id)
{
    for(int i = 0; i < partner->pCount; i++)
    {
        if(partner->perfect[i] == id)
        {
            return i;
        }
    }
```

```
        //返回一个非常大的值，意味着根本排不上对
        return 0x7FFFFFFF;
    }
```

按照“稳定婚姻问题”的要求，这个函数应该总是能够得到 ID 指定的异性舞伴在 partner 的偏爱列表中的位置，因为每个 partner 的偏爱列表包含所有异性舞伴。但是当题目有特殊需求时，partner 的偏爱列表可能只有部分异性舞伴。比如 partner 非常恨一个人，他们绝对不能成为舞伴，那么 partner 的偏爱列表肯定不会包含这个人。考虑到算法的通用性，GetPerfectPosition()函数默认返回一个非常大的数，返回这个数这意味着 ID 指定的异性舞伴在 partner 的偏爱列表中根本没有位置（非常恨），根据算法的规则，partner 最不喜欢的异性舞伴的位置都比 id 指定的异性舞伴位置靠前。这也是算法一致性处理的一个技巧，GetPerfectPosition()函数当然可以设计成返回–1 表示 ID 指定的异性舞伴不在 partner 的偏爱列表中，但是大家想一想，算法中是不是要对这个返回值做特殊处理？原来代码中判断位置关系的一行代码处理：

```
if(GetPerfectPosition(&girls[gid], bpid) > GetPerfectPosition(&girls[gid], bid))
```

就会变得非常繁琐，让我们看看会是什么情况：

```
if((GetPerfectPosition(&girls[gid], bpid) == -1)
    && (GetPerfectPosition(&girls[gid], bid) == -1))
{
    //当前舞伴 bpid 和 bid 都不在女孩的喜欢列表中，太糟糕了
    ...
}
else if(GetPerfectPosition(&girls[gid], bpid) == -1)
{
    //当前舞伴 bpid 不在女孩的喜欢列表中，bid 有机会
    ...
}
else if(GetPerfectPosition(&girls[gid], bid) == -1)
{
    //bid 不在女孩的喜欢列表中，当前舞伴 bpid 维持原状
    ...
}
else if(GetPerfectPosition(&girls[gid], bpid) > GetPerfectPosition(&girls[gid], bid))
{
    //女孩喜欢 bid 胜过其当前舞伴 bpid
    ...
}
else
{
    //女孩喜欢当前舞伴 bpid 胜过 bid
    ...
}
```

这是我最不喜欢的代码逻辑，真的，太糟糕了。可见，这个小小的技巧为代码的逻辑处理带来了极大的好处。类似的技巧被广泛应用，在排序算法中经常使用“哨兵”位，避免每次都要判断是否比较完全部元素。面向对象技术中常用的“Dummy Object”技术，也是类似的思想。

Gale-Shapley 算法原来如此简单，你是不是为沙普利能获得诺贝尔奖愤愤不平？其实不然，算法原理的简单并不等于其解决的问题也简单，本书介绍的很多算法都是如此，小算法解决大问题。

7.2.2　改进优化：空间换时间

Gale_Shapley()函数给出的算法还有点问题，主要是 GetPerfectPosition()函数的策略，这个函数每次都要遍历 partner 的偏爱舞伴列表才能确定 bid 的位置，很可能导致理论上时间复杂度为 $O(n^2)$的算法在实际实现时变成 $O(n^3)$的时间复杂度。为了避免算法在多轮选择过程中频繁遍历每个 partner 的偏爱舞伴列表，需要对 partner 到底更偏爱哪个舞伴的判断策略进行改进。

改进的原则就是“以空间换时间”。简单来讲，以空间换时间的方法就是用一张事先初始化好的表存储这些位置关系，在使用个过程中，以 $O(1)$时间复杂度的方式直接查表确定偏爱舞伴的关系。这样的表可以是线性表，也可以是哈希表这样的映射表。对于这个问题，我们选择使用二维表来存储这些位置关系。假设存在二维表 priority[n][n]，我们用 priority[w][m]表示 m 在 w 的偏爱列表中的位置，这个值越小，表示 m 在 w 的偏爱列表中的位置越靠前。在算法开始之前，首先初始化这个关系表：

```
for(int w = 0; w < UNIT_COUNT; w++)
{
    //初始化成最大值，原理同上
    for(int j = 0; j < UNIT_COUNT; j++)
    {
        priority[w][j] = 0x7FFFFFFF;
    }
    //给偏爱舞伴指定位置关系
    int pos = 0;
    for(int m = 0; m < girls[w].pCount; m++)
    {
        priority[w][girls[w].perfect[m]] = pos++;
    }
}
```

最后，将对 GetPerfectPosition()函数的调用替换成查表：

```
if(priority[gid][bpid] > priority[gid][bid])
```

对于一些在算法执行过程中不会发生变化的静态数据，如果算法执行过程中需要反复读取这些数据，并且读取操作存在一定时间开销的场合，比较适合使用这种“以空间换时间”的策略。用合理的方式组织这些数据，使得数据能够在 $O(1)$时间复杂度内实现是这种策略的关键。对本问题应用“以空间换时间”的策略，需要在算法开始的准备阶段初始化好 priority 二维表，这需要一些额外的开销，但是相对于 n^2 次查询节省的时间来说，这点开销是能够容忍的。

“以空间换时间”也是算法设计常用的技巧，在很多算法中都有应用。比如本书第 15 章介绍的快速傅里叶变换算法，经过蝶形变换后每个点的数据位置都发生了变化，需要将这些点的位置

还原。可以利用一个二重循环将这些错位的数据还原，也可以利用蝶形变换的位置变换关系表，采用查表的方式将两个错位的数据交换位置，后者采用的就是“以空间换时间”的策略。

7.3 有多少稳定匹配

当参加舞会的男孩和女孩按照一定的顺序排好队，位置固定之后，使用 Gale-Shapley 算法能够得到一个确定的稳定匹配结果。但是对这群男孩和女孩来说，稳定匹配的结果肯定不是唯一的，其实只要将计算策略从“男士优先”转换成“女士优先”，就可以得到另外一个完全不同的稳定匹配结果。同样，调整一下男孩们的位置顺序，比如让最后一个男孩排在第一的位置，让他第一个邀请女孩，则 Gale-Shapley 算法也可以得到一个完全不同的稳定匹配结果。

很显然，对于任意情况下的 n 个男孩和 n 个女孩来说，肯定有多个稳定匹配，那么，到底有多少个稳定匹配？稳定匹配首先必须是完美匹配，而且稳定匹配的个数小于或等于完美匹配，所以，我们可以先从理论上计算一下完美匹配的数量，估算一下问题的规模，然后再决定是否能用算法找出全部的稳定匹配。从理论上分析，只要每个人的偏爱列表都包含全部异性舞伴，那么完美匹配的个数就可以通过公式计算出来。首先，假设男孩们已经排好了队，准备按照顺序邀请女孩跳舞，在不考虑稳定匹配的情况下，每个男孩选择一个女孩之后，还没有舞伴的女孩的总数就减 1，剩下的男生的可选范围就变小了。第一个男孩选择的可能情况是 C_n^1，第二个男孩可能的选择就只有 C_{n-1}^1 种。以此类推，可以计算出完美匹配的可能情况是 $M = C_n^1\ C_{n-1}^1\ C_{n-2}^1 \dots C_1^1 = n!$种。如果仅仅从排列组合问题的角度考虑舞伴问题，随着男孩们的顺序变化，这个数字会成倍增加。那么男孩们有多少种顺序变化呢？n 个男孩全排列，结果也是 $n!(P_n^n)$种变化，因此最终的结果应该是$(n!)^2$。但是舞伴问题并不是单纯的排列组合问题，因为这些男孩和女孩之间通过各自的偏爱列表建立了某种联系，这使得一些组合结果实际上是没有意义的重复。举个例子说明一下，假如 m 在第一轮选择，他选择 w 作为舞伴，m'在第二轮选择，他选择 w'作为舞伴。现在转换一下选择顺序，改为 m'在第一轮选择，但是 m'的偏爱列表中，w'排在前面，于是 m'仍然选择 w'作为舞伴，m 也只能选择 w 作为舞伴，虽然选择的顺序变了，但是结果和前一次一样。

由此看来，虽然对男孩的选择顺序进行全排列有 $n!$种可能，但是这 $n!$种选择顺序最终得到的匹配结果都只是 $n!$种结果的重复出现，实际的完美匹配只有 $n!$种。接下来我们要给出的穷举算法也验证了这一点，对于 3 个男孩和 3 个女孩的情况，穷举算法得到了 36（3!×3!=36）个完美匹配结果，排除掉重复结果后得到 6（3×2×1=6）个结果。对于 4 个男孩和 4 个女孩的情况，穷举算法得到了 576（4!×4!=576）个完美匹配结果，排除掉重复结果后得到 24（4×3×2×1=24）个结果。

7.3.1 穷举所有的完美匹配

如果想知道到底有多少个稳定匹配，首先要知道有多少个完美匹配。具体的方法就是使用穷举的方法找到全部的完美匹配，然后根据条件将包含不稳定因素的完美匹配过滤掉，剩下的就是稳定匹配。遵循这个原则，我们先来研究一下穷举所有完美匹配的算法。

穷举算法的数据结构定义仍然沿用 7.2.1 节算法实现中使用的 PARTNER 定义，只是 next 属性用不上。穷举的方法就是每次为一个男孩选择一个舞伴，选择的方法就是从男孩的偏爱列表中找一个还没有舞伴的女孩，确定为这个男孩的舞伴，同时将男孩和女孩对应的 PARTNER 定义中的 current 属性指向对方。判断一个女孩是否已经有舞伴的方法就是判断她的 current 属性是否是 –1，如果不是–1 就说明这个女孩已经有舞伴了。按照男孩的顺序逐个为他们选择舞伴，当最后一个男孩也确定了舞伴之后，就得到了一个完美匹配，可以打印这个结果，用于检查是否正确。

SearchStableMatch()函数是搜索算法的核心，采用递归方式实现，每次为一个男孩选择舞伴。index 参数是男孩按照顺序的编号，从 0 开始编号，刚好对应 boys 数组的下标，简化了代码实现。当 index 等于 UNIT_COUNT（男孩的个数）时，表示已经为所有男孩找到了舞伴，如果算法没有错误，这应该就是一个完美匹配。算法的主体就是遍历 index 对应的男孩的偏爱列表，从列表中找到一个还没有舞伴并且也喜欢自己的女孩作为舞伴，互相设置 current 属性。需要注意的是，算法主体包含一个回溯处理，当某一级搜索结束后，要重置相关男孩和女孩的舞伴关系，以便后序的递归搜索能够正常进行。具体代码可看 SearchStableMatch()函数的 for 循环主体部分。

```
void SearchStableMatch(int index, PARTNER *boys, PARTNER *girls)
{
    if(index == UNIT_COUNT)
    {
        if(IsStableMatch(boys, girls))
        {
            PrintResult(boys, girls, UNIT_COUNT);
        }
        return;
    }

    for(int i = 0; i < boys[index].pCount; i++)
    {
        int gid = boys[index].perfect[i];

        if(!IsPartnerAssigned(&girls[gid]) && IsFavoritePartner(&girls[gid], index))
        {
            boys[index].current = gid;
            girls[gid].current = index;
            SearchStableMatch(index + 1, boys, girls);
            boys[index].current = -1;
            girls[gid].current = -1;
        }
    }
}
```

7.3.2　不稳定因素的判断算法

7.1.1 节给出了完美匹配中不稳定因素的定义，当一个男孩和一个女孩同时有比他们当前舞伴更“强烈的”愿望结为舞伴的时候，他们就倾向于与各自的舞伴分开，然后结合在一起成为舞伴。不稳定因素的判断算法就是在一个完美匹配中找出图 7-1 所示的情况，这种情况有两个特征：

首先，男孩的当前舞伴不是他的偏爱列表中排在第一位的女孩，也就是说，男孩更偏爱其他女孩胜过自己当前的舞伴；其次，男孩更偏爱的那个（或那几个女孩中的一个）女孩刚好也喜欢这个男孩胜过自己当前的舞伴。

于是，不稳定因素的判断算法就呼之欲出了，重点就是上述两个特征的识别。判断一个完美匹配是否是稳定匹配的算法流程如下。

(1) 找出这个男孩的当前舞伴在男孩的偏爱列表中的位置，如果当前舞伴排在偏爱列表的第一位，则表示这个男孩不存在不稳定因素的可能，转步骤(4)。如果当前舞伴不是男孩偏爱列表的第一位，则转到步骤(2)。

(2) 男孩的偏爱列表中如果还有排在当前舞伴之前但还没有进行判断处理的女孩，则转步骤(3)，否则转步骤(4)。

(3) 找到女孩的当前舞伴在女孩的偏爱列表中的位置和当前处理的男孩在女孩的偏爱列表中的位置，如果女孩当前舞伴的位置比当前处理的男孩的位置靠前，则表示对该女孩不存在不稳定因素，转步骤(2)。如果当前处理的男孩的位置比女孩当前舞伴的位置靠前，则表示存在不稳定因素，直接转步骤(6)。

(4) 如果对全部男孩判断完毕，转步骤(5)。否则，继续对下一个男孩进行不稳定因素判断，转步骤(1)。

(5) 结束，没有找到不稳定因素。

(6) 结束，找到不稳定因素，此完美匹配不是稳定匹配。

根据以上算法流程，我们给出判断稳定匹配的算法实现，如 `IsStableMatch()`函数所示，非常简单，相关的注释和以上算法流程的表述都能对上，此处就不再过多解释。

```
bool IsStableMatch(PARTNER *boys, PARTNER *girls)
{
    for(int i = 0; i < UNIT_COUNT; i++)
    {
        //找到男孩当前舞伴在自己的偏好列表中的位置
        int gpos = GetPerfectPosition(&boys[i], boys[i].current);
        //在 position 位置之前的舞伴，男孩喜欢她们胜过 current
        for(int k = 0; k < gpos; k++)
        {
            int gid = boys[i].perfect[k];
            //找到男孩在这个女孩的偏好列表中的位置
            int bpos = GetPerfectPosition(&girls[gid], i);
            //找到女孩的当前舞伴在这个女孩的偏好列表中的位置
            int cpos = GetPerfectPosition(&girls[gid], girls[gid].current);
            if(bpos < cpos)
            {
                //女孩也是喜欢这个男孩胜过喜欢自己当前的舞伴，这是不稳定因素
                return false;
            }
        }
    }
```

```
    return true;
}
```

7.3.3 穷举的结果

至此，我们有了穷举法搜索全部稳定匹配结果的算法，来看看结果吧。假设有以下男孩和女孩的数据，冒号后是对应男孩和女孩的偏爱列表。

男孩

```
Albert:  Laura, Nancy, Marcy
Brad:  Marcy, Nancy, Laura
Chuck:  Laura, Marcy, Nancy
```

女孩

```
Laura:  Chuck, Albert, Brad
Marcy:  Albert, Chuck, Brad
Nancy:  Brad, Albert, Chuck
```

应用算法搜索后得到以下结果：

```
Albert[1] <---> Nancy[1]
Brad[0] <---> Marcy[2]
Chuck[0] <---> Laura[0]

Albert[2] <---> Marcy[0]
Brad[1] <---> Nancy[0]
Chuck[0] <---> Laura[0]

Total Matchs : 6, Stable Matchs : 2
```

看来，有两个稳定匹配的结果，用 7.2.1 节给出的 Gale-Shapley 算法得到的只是前一个稳定匹配的结果。参考资料[6]给出了一个有意思的结论，就是稳定匹配的个数总是 2 的整数幂，有兴趣的读者可阅读一下该资料，看看这个结论的来龙去脉。另外，这个资料还给出了只有一种稳定匹配结果的情况，即所有的女孩的偏爱列表都完全一样的时候，无论男孩们的偏爱列表如何选择，最终都只有一种稳定匹配结果，有兴趣的读者也可以自己研究研究。

7.4 二部图与二分匹配

之前讨论稳定匹配问题的时候，我们把完美匹配定义为每个男人和女人都属于匹配中的某个对，并不是很直观，现在我们准备用图的术语更一般地表达完美匹配的概念。首先介绍一下二部图，二部图 $G=(V,E)$是这样的一个图，它的顶点集合 V 可以划分为 X 和 Y 两个集合，它的边集合 E 中的每条边都有一个端点在 X 集合，另一个端点在 Y 集合。图 7-2 就是一个二部图。

现在给出针对二部图的匹配的定义，给定一个二部图 $G=(V,E)$的子图 M，如果 M 的边集中任意两条边都不依附于同一个顶点，则称 M 是一个匹配。简单地说，图 7-2 中 x_2、x_3、x_4 等点都有

多条边与之连接，也就是说有多个边依附于这些点，因此图 7-2 所示的二部图不是一个匹配。现在考虑删除一些边，最终得到如图 7-3 所示的一个 G 的子图。该子图中没有任何边同时连接 X 或 Y 中的同一个顶点，因此这是一个匹配。

如果 G 的一系列子图 $M_0,M_1,\cdots,M_n$ 都是匹配，那么包含边数最多的那个匹配就是图 G 的最大匹配。如果一个最大匹配中所有的点都有边与之相连，没有未覆盖点，则这个最大匹配就是完美匹配。未覆盖点的定义是：图 G 的一个顶点 V_i，如果 V_i 不与任何一条属于匹配 M 的边相连，则成 V_i 是一个未覆盖点。图 7-3 就是一个完美匹配。

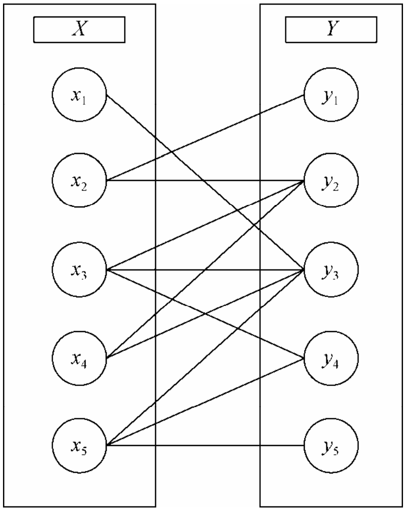


图 7-2　简单的二部图

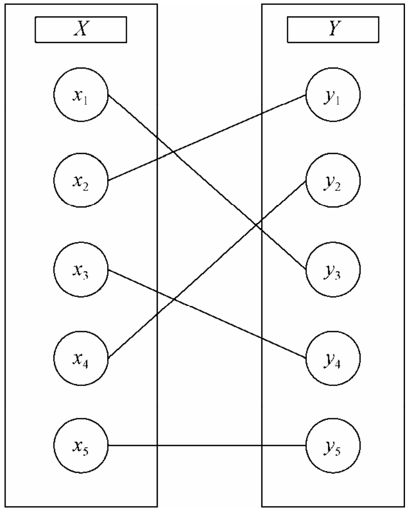


图 7-3　一个完美匹配的二部图

根据以上定义，如果 G 的一个匹配 M 是最大匹配，并且没有未覆盖点，则这个匹配就是完美匹配。可见，图 G 的匹配和完美匹配正好就是之前介绍的“稳定婚姻问题”中的匹配和完美匹配。用图论的方法寻找完美匹配，需要首先找到最大匹配，当二部图中两个顶点集合中的顶点个数相等时，这个最大匹配同时也是完美匹配。求二部图的最大匹配可以使用最大流（maximal flow）或匈牙利算法（Hungarian algorithm），接下来我们就来介绍匈牙利算法。

7.4.1　最大匹配与匈牙利算法

寻找二部图最大匹配的匈牙利数学家埃德蒙德斯（Edmonds）在 1965 年提出的一个简化的最大流算法。该算法根据二部图匹配这个问题的特点将最大流算法进行了简化，提高了效率。普通的最大流算法一般都是基于带权网络模型的，二部图匹配问题不需要区分图中的源点和汇点，也不关心边的方向，因此不需要复杂的网络图模型，这就是匈牙利算法简化的原因。正是因为这个原因，匈牙利算法成为一种很简单的二分匹配算法，其基本流程是：

```
将图 G 最大匹配初始化为空
while(从 Xi 点开始在图 G 中找到新的增广路径)
{
    将增广路径加入到最大匹配中;
}
输出图 G 的最大匹配;
```

根据匈牙利算法的流程看，寻找图 G 中的增广路径（Augment Path）是匈牙利算法的关键。先来看看什么是增广路径，二部图中的增广路径具有以下性质：

- 路径中边的条数是奇数；
- 路径的起点在二部图的左半边，终点在二部图的右半边；
- 路径上的点一个在左半边，一个在右半边，交替出现，整条路径上没有重复的点；
- 只有路径的起点和终点都是未覆盖的点，路径上其他的点都已经配对；
- 对路径上的边按照顺序编号，所有奇数编号的边都不在已知的匹配中，所有偶数编号的边都在已知的匹配中；
- 对增广路径进行“取反”操作，新的匹配数就比已知匹配数增加一个，也就是说，可以得到一个更大的匹配。

所谓的增广路径取反操作，就是把增广路径上奇数编号的边加入到已知匹配中，并把增广路径上偶数编号的边从已知匹配中删除。每做一次“取反”操作，得到的匹配就比原匹配多一个。匈牙利算法的思路就是不停地寻找增广路径，增加匹配的个数，当不能再找到增广路径时，算法就结束了，得到的匹配就是最大匹配。

增广路径的起点总是在二部图的左边，因此寻找增广路径的算法总是从一侧的顶点开始，逐个顶点搜索。从 X_i 顶点开始搜索增广路径的流程如下：

```
while(从 Xi 的邻接表中找到下一个关联顶点 Yj)
{
    if(顶点 Yj 不在增广路径上)
    {
        将 Yj 加入增广路径;
        if(Yj 是未覆盖点或者从与 Yj 相关连的顶点（Xk）能找到增广路径)
        {
            将 Yj 的关联顶点修改为 Xi;
            从顶点 Xi 开始有增广路径，返回 true;
        }
    }
    从顶点 Xi 开始没有增广路径，返回 false;
}
```

在这个算法流程中，“从与 Y_j 相关连的顶点（X_k）能找到增广路径”这一步体现的是一个递归过程。因为如果之前的搜索已经将 Y_j 加入到增广路径中，说明 Y_j 在 X 集合中一定有一个关联点，我们假设 Y_j 在 X 集合中的这个关联点是 X_k，所以要从 X_k 开始继续寻找增广路径。当从 X_k 开始的递归搜索完成后，通过“将 Y_j 的关联顶点修改为 X_i”这一步操作，将其与 X_i 连在一起，形成一条更长的增广路径。

到现在为止，匈牙利算法的流程已经很清楚了，现在我们来给出实现代码。首先定义求最大匹配的数据结构，这个数据结构要能表示二部图的边的关系，还要能体现最终的增广路径结果，我们给出如下定义：

```
typedef struct tagMaxMatch
{
    int edge[UNIT_COUNT][UNIT_COUNT];
    bool on_path[UNIT_COUNT];
    int path[UNIT_COUNT];
    int max_match;
}GRAPH_MATCH;
```

edge 是顶点与边的关系表，用来表示二部图，on_path 用来表示顶点 Y_j 是否已经在当前搜索过程中形成的增广路径上了，path 是当前找到的增广路径，max_match 是当前增广路径中边的条数，当算法结束时，如果 max_match 不等于顶点个数，说明有顶点不在最大增广路径上，也就是说，找不到能覆盖所有点的增广路径，此二部图没有最大匹配。从 X_i 寻找增广路径的算法实现如下：

```
bool FindAugmentPath(GRAPH_MATCH *match, int xi)
{
    for(int yj = 0; yj < UNIT_COUNT; yj++)
    {
        if((match->edge[xi][yj] == 1) && !match->on_path[yj])
        {
            match->on_path[yj] = true;
            if( (match->path[yj] == -1)
                || FindAugmentPath(match, match->path[yj]) )
            {
                match->path[yj] = xi;
                return true;
            }
        }
    }

    return false;
}
```

算法实现基本上是按照之前的算法流程实现的，不需要做特别说明，唯一需要注意的是 path 中存放增广路径的方式。读者可能已经注意到了，存放的方式是以 Y 集合中的顶点为索引存放，其值是对应的关联顶点在 X 集合中的索引。搜索是按照 X 集合中的顶点索引进行的，增广路径以 Y 集合中的顶点为索引存储，关系是反的。输出结果的时候，需要结合 Y 集合中的顶点索引输出，如果需要以 X 集合的顺序输出结果，需要反向转换，转换的方法非常简单：

```
int path[UNIT_COUNT] = { 0 };

for(int i = 0; i < match->max_match; i++)
{
    path[match->path[i]] = i;
}
```

转换后 path 中就是以 X 集合的顺序存放的结果。

结合之前给出的匈牙利算法基本流程，最后给出匈牙利算法的入口函数实现：

```
bool Hungary_Match(GRAPH_MATCH *match)
{
    for(int xi = 0; xi < UNIT_COUNT; xi++)
    {
        if(FindAugmentPath(match, xi))
        {
            match->max_match++;
        }

        ClearOnPathSign(match);
    }
    return (match->max_match == UNIT_COUNT);
}
```

每完成一个顶点的搜索，需要重置 Y 集合中相关顶点的 on_path 标志，ClearOnPathSign()函数就负责干这个事情。

我们用图 7-2 中的二部图数据初始化 GRAPH_MATCH 中的顶点关系表 edge，然后调用 Hungary_Match()函数得到一组匹配：

```
X1<--->Y3
X2<--->Y1
X3<--->Y4
X4<--->Y2
X5<--->Y5
```

结果与图 7-3 一致，因为这个最大匹配没有未覆盖点，所以是完美匹配。

匈牙利算法的实现以顶点集合 V 为基础，每次 X 集合中选一个顶点 X_i 做增广路径的起点搜索增广路径。搜索增广路径需要遍历边集 E 内的所有边，遍历方法可以采用深度优先遍历（DFS），也可以采用广度优先遍历（BFS），无论什么方法，其时间复杂度都是 $O(E)$。匈牙利算法每个顶点 V_i 只能选择一次，因此算法的整体时间复杂度是 $O(V*E)$，总的来说，是一个相当高效的算法。除了匈牙利算法，求二部图的最大匹配还可以使用 Hopcroft-Karp 算法。Hopcroft-Karp 算法是由 Hopcroft 和 Karp 在 1972 年提出的一种算法，也是最大流算法的一种改进算法。算法的基本思想就是在每次搜索增广路径的时候不是只找一条增广路径，而是同时找几条互不相交的增广路径，形成最大增广路径集合，然后沿着集合中的几条增广路径同时扩大增广路径长度。通过进一步的分析，Hopcroft-Karp 算法的时间复杂度可以达到 $O(\mathrm{sqrt}(V)*E)$，也非常高效。

7.4.2　带权匹配与 Kuhn-Munkres 算法

上一节我们介绍了二部图的最大匹配算法，用匈牙利算法寻找最大匹配，不要求每个个体给出的偏爱列表包含全部异性成员。比如在舞伴问题中，如果 Albert 非常讨厌 Marcy，那么 Albert 的偏爱列表无论如何也不会包含 Marcy。在这一节，我们让舞伴问题再复杂一点，于是为舞伴

问题引入带权优先表的概念，为每一个配对指定一个权重，表明我们更希望哪一对成为舞伴。通过控制每一对舞伴关系的权重，使得最后的完美匹配结果中有尽量多的舞伴是我们所期望的配对关系。

这个问题变得有点像最优解问题了，一提到与图有关的最优解问题，你会想到穷举法。穷举所有的完美匹配，然后计算每个完美匹配中各边的权重之和，取权重之和最大的一个作为最后的结果。这是一种解决方案，但是穷举法虽然是万能方法，但是不到万不得已最好不要用穷举法。仔细思考一下，其实这个问题已经演化成了求解二部图的带权匹配问题，所谓二部图的带权匹配其实就是求出一个匹配集合，使得集合中各边的权值之和最大或最小。对于本问题，给每一个配对（图中的边）指定一个权重之后问题就变成了求二部图的带权最大匹配问题。

通过之前对 Gale-Shapley 算法和匈牙利算法的介绍，我们已经了解了完美匹配、稳定匹配和最大匹配这些概念，那么带权匹配和之前的这些概念是什么关系呢？答案是没有半毛钱关系，至少和完美匹配与最大匹配之间不存在包含或等于关系。二部图的最大权或最小权匹配，只是要求得到的一个匹配中各边的权值之和最大或最小，并不要求这个匹配是完美匹配或最大匹配。如果这个权值最大（或最小）的匹配同时又是完美匹配，则这样的结果就被称为最佳匹配。本节我们要介绍的 Kuhn-Munkres 算法是求最大权或最小权匹配的算法，如果期望 Kuhn-Munkres 算法得到的结果同时是一个完美匹配（最佳匹配），那么要求算法运行的数据必须存在完美匹配（比如两个顶点集合的顶点个数必须相等之类的条件）。很多同学会忽略这一点，以为 Kuhn-Munkres 算法可以在任何情况下得到带权的最大匹配，这个理解是错误的。

Kuhn-Munkres 算法，也称 KM 算法，是 Kuhn 和 Munkres 二人在 1955 ~ 1957 年各自独立提出的一种算法，是一种求解最大最小权匹配问题的经典算法。最初的 Kuhn-Munkres 算法以矩阵为基础结构，但是 Edmonds 在 1965 年发布了匈牙利算法之后，Kuhn-Munkres 算法也基于匈牙利算法进行了改进。当给定的二部图存在完美匹配的情况下，Kuhn-Munkres 算法通过给每个顶点设置一个标号（叫作顶标）的方式把求最大权匹配的问题转化为求完美匹配的问题的，最终得到一个最大权完美匹配。那么这个转换是如何实现的呢？这就需要分析一下 Kuhn-Munkres 算法的原理了。

我们假设二部图中 X 顶点集合中每个顶点 X_i 的顶标是 $A[i]$，Y 顶点集合中每个顶点 Y_i 的顶标是 $B[i]$，顶点 X_i 和 Y_i 之间的边的权重是 weight$[i][j]$，则 Kuhn-Munkres 算法的原理就是基于以下定理：

若由二部图中所有满足 $A[i]+B[j]$ = weight$[i,j]$ 的边 (X_i,Y_j) 构成的子图（称作相等子图）有完美匹配，那么这个完美匹配就是二分图的最大权匹配。

现在明白转换原理了吧，就是先找出问题对应的相等子图，然后求相等子图的完美匹配即可。现在的问题是，这个定义成立吗？答案是只要在算法过程中始终满足“$A[i]+B[j] \geqslant$ weight$[i,j]$”这个条件，这个定理就成立。因为对于二分图的任意一个匹配，如果这个匹配是相等子图的匹配，那么它的边权重之和等于所有顶点的顶标之和（显然这是最大的）；如果这个匹配不是相等子图

的匹配（它的某些边不属于相等子图），那么它的边权重之和小于所有顶点的顶标和。所以只要始终满足“$A[i]+B[j] \geqslant \text{weight}[i,j]$”条件的相等子图的完备匹配一定是二分图的最大权匹配。

根据以上分析可知，Kuhn-Munkres 算法的实现流程大致如下所示。

(1) 初始化各个顶点的顶标值；

(2) 找出符合“$A[i]+B[j] = \text{weight}[i,j]$”条件的边构成相等子图，使用匈牙利算法寻找相等子图的完美匹配；

(3) 如果找到相等子图的完美匹配，则算法结束，否则调整相关顶点的顶标值；

(4) 重复步骤(2)(3)，直到找到完美匹配为止。

第(1)步初始化顶点顶标值可采用式(7-1)计算：

$$\begin{cases} A[x_i] = \max\{\text{weight}[x_i][y_0], \text{weight}[x_i][y_1], \cdots, \text{weight}[x_i][y_n]\}, x_i \in X \\ B[y_i] = 0, y_i \in Y \end{cases} \tag{7-1}$$

因为 $A[i]$总是取与之相邻的边中最大的权值作为初始值，因此初始阶段能保证满足“$A[i]+B[j] \geqslant \text{weight}[i,j]$”条件。如果在第(2)步的相等子图中没有找到完美匹配，说明相等子图中某个顶点出发的增广路径不能覆盖所有顶点。此时需要调整各个顶点的顶标值，然后重新在相等子图中寻找完美匹配。调整顶标的目的是为了扩大相等子图，使得更多的边进入相等子图，并最终能够找到一个完美匹配。设当前增广路径上所有属于 X 集合的顶点构成一个子集 S，所有属于 Y 集合的顶点构成一个子集 T，dx 为顶标调整的变化量，则 dx 可采用式(7-2)给出的方法计算：

$$\mathrm{d}x = \min\{A[x_i] + B[y_j] - \text{weight}[x_i][y_j], x_i \in S, y_j \notin T\} \tag{7-2}$$

由 dx 的计算公式可知，如果把 S 集合中所有顶点的顶标都减少 dx，一定会有一条一端在 S 中，另一端不在 T 中的边因满足“$A[i]+B[j] = \text{weight}[i,j]$”的条件而进入相等子图，这就扩大了相等子图。对 S 集合中所有顶点的顶标都减少 dx 之后，为了使原来已经在相等子图中的边继续留在相等子图中，需要将 T 集合中所有顶点的顶标值增加 dx，使 $A[i]+B[j]$之和不变。

现在总结一下顶标调整的方法，首先采用式(7-2)计算出调整变化量 dx，然后将 S 集合中所有顶点的顶标值减少 dx，同时将 T 集合中所有顶点的顶标值增加 dx，这样的调整，对整个图上的所有顶点会产生如下四种结果。

- ❑ 对于两端点都在当前相等子图的增广路径上的边（x_i, y_j），其顶标值 $A[i]+B[j]$的和没有变化。也就是说，原来属于相等子图的边，调整后仍然属于相等子图。
- ❑ 对于两端点都不在当前相等子图的增广路径上的边（x_i, y_j），其顶标值 $A[i]$和 $B[j]$的值没有变化。也就是说，此边与相等子图的隶属关系没有变化，原来属于相等子图的边现在仍然属于相等子图，原来不属于相等子图的边现在仍然不属于相等子图。
- ❑ 对于 x_i 在当前相等子图的增广路径上，y_j 不在当前相等子图的增广路径上的边（x_i, y_j），其顶标值 $A[i]+B[j]$的和略有减小，原来不属于相等子图，现在有可能属于相等子图，使得相等子图有机会得到扩大。

- 如果 x_i 不在当前相等子图的增广路径上，y_j 在当前相等子图的增广路径上的边（x_i, y_j），其顶标值 $A[i]+B[j]$ 的和略有增加，原来不属于相等子图，现在仍不属于相等子图。

由此可见，每次调整顶标，都能在图的基本状态保持不变的情况下扩大相等子图，使得相等子图有机会找到一个完美匹配，这就是顶标值调整在算法中的意义所在。

现在，我们就结合一个带权最大匹配问题的实例，给出这个算法在实际应用中的一个实现。问题是这样的，某公司有 5 名技术工人，他们都可以完成公司流程中的 5 种工作，但是每个工人的技术侧重点不同，熟练程度也不同，因此他们完成同样的工作所产生的经济效益也不相同。如果用 0 ~ 5 范围内的值对每个工人完成每种工作所产生的经济效益进行评价，可得到如表 7-1 所示的经济效益矩阵。假如你是这家公司的负责人，你需要找到一种工人和工作之间的匹配关系，使得这种匹配关系能产生的经济效益最大。根据之前对 Kuhn-Munkres 算法的分析，我们针对这个问题设计了 KM_MATCH 匹配数据结构，如下所示：

```
typedef struct tagKmMatch
{
    int edge[UNIT_COUNT][UNIT_COUNT]; //Xi 与 Yj 对应的边的权重
    bool sub_map[UNIT_COUNT][UNIT_COUNT];// 二分图的相等子图, sub_map[i][j] = 1 代表 Xi 与 Yj 有边
    bool x_on_path[UNIT_COUNT]; // 标记在一次寻找增广路径的过程中，Xi 是否在增广路径上
    bool y_on_path[UNIT_COUNT]; // 标记在一次寻找增广路径的过程中，Yi 是否在增广路径上
    int path[UNIT_COUNT]; // 匹配信息，其中 i 为 Y 中的顶点标号，path[i]为 X 中顶点标号
}KM_MATCH;
```

相对于匈牙利算法中的 GRAPH_MATCH 定义，KM_MATCH 的主要变化是增加了 sub_map 作为相等子图定义和标识 y_i 是否在增广路径上的 y_on_path 标识。相对于我们前面对 Kuhn-Munkres 算法的分析，edge 对应于边的权重表 weight，sub_map 对应于算法执行过程中的相等子图，x_on_path 和 y_on_path 分别用于标识 X 集合和 Y 集合中的顶点是否属于增广路径上的 S 集合和 T 集合，path 就是最后匹配的结果。

表7-1　不同工人完成不同工作的经济效益

	工作1	工作2	工作3	工作4	工作5
工人1	3	5	5	4	1
工人2	2	2	0	2	2
工人3	2	4	4	1	0
工人4	0	1	1	0	0
工人5	1	2	1	3	3

下面给出 Kuhn-Munkres 算法的具体实现代码，Kuhn_Munkres_Match()函数虽然很长，但是并不难理解，因为这个代码是严格按照之前给出的 Kuhn-Munkres 算法的流程实现的。包括顶标的初始化、使用匈牙利算法求完美匹配和顶标调整在内的三个主要算法步骤在 Kuhn_Munkres_Match()函数中都得到体现，并且界定非常清晰。其中寻找增广路径的 FindAugmentPath()函数与之前介绍匈牙利算法时给出的 FindAugmentPath()函数实现非常类似，区别就是使用 sub_map 而不是直接使用 edge，并且额外记录了 x_on_path 标识。ResetMatchPath()函数负责每次开始寻找相等子图之前

清除上一次搜寻产生的临时增广路径，ClearOnPathSign()函数负责在每次搜寻增广路径之前清除顶点是否属于 S 集合或 T 集合的标识，大家可以从本书的配套代码中找到此函数的代码。

```
bool Kuhn_Munkres_Match(KM_MATCH *km)
{
    int i, j;
    int A[UNIT_COUNT], B[UNIT_COUNT];
    // 初始化 Xi 与 Yi 的顶标
    for(i = 0; i < UNIT_COUNT; i++)
    {
        B[i] = 0;
        A[i] = -INFINITE;
        for(j = 0; j < UNIT_COUNT; j++)
        {
            A[i] = std::max(A[i], km->edge[i][j]);
        }
    }
    while(true)
    {
        // 初始化带权二分图的相等子图
        for(i = 0; i < UNIT_COUNT; i++)
        {
            for(j = 0; j < UNIT_COUNT; j++)
            {
                km->sub_map[i][j] = ((A[i]+B[j]) == km->edge[i][j]);
            }
        }
        //使用匈牙利算法寻找相等子图的完备匹配
        int match = 0;
        ResetMatchPath(km);
        for(int xi = 0; xi < UNIT_COUNT; xi++)
        {
            ClearOnPathSign(km);
            if(FindAugmentPath(km, xi))
                match++;
            else
            {
                km->x_on_path[xi] = true;
                break;
            }
        }
        //如果找到完备匹配就返回结果
        if(match == UNIT_COUNT)
        {
            return true;
        }
        //调整顶标，继续算法
        int dx = INFINITE;
        for(i = 0; i < UNIT_COUNT; i++)
        {
            if(km->x_on_path[i])
            {
                for(j = 0; j < UNIT_COUNT; j++)
```

```
                {
                    if(!km->y_on_path[j])
                        dx = std::min(dx, A[i] + B[j] - km->edge[i][j]);
                }
            }
        }
        for(i = 0; i < UNIT_COUNT; i++)
        {
            if(km->x_on_path[i])
                A[i] -= dx;
            if(km->y_on_path[i])
                B[i] += dx;
        }
    }

    return false;
}
```

根据表 7-1 提供的数据初始化 KM_MATCH 数据结构，然后调用 Kuhn_Munkres_Match()函数，得到一个最大权匹配的结果，因为原数据存在完美匹配，因此这个结果就是最佳匹配结果：

工人 1 分配 工作 3（经济效益评价是 5）
工人 2 分配 工作 1（经济效益评价是 2）
工人 3 分配 工作 2（经济效益评价是 4）
工人 4 分配 工作 5（经济效益评价是 0）
工人 5 分配 工作 4（经济效益评价是 3）

最后获得最大经济效益评价是 14。需要说明的是，对于同一个问题，其最大权匹配的结果可能不唯一，也就是说，存在多个匹配的权重之和同为最大值的情况。Kuhn-Munkres 算法可以找出其中的一个，但是无法找到全部匹配结果。

7.5 总结

各种匹配问题可不是仅仅用来娱乐的算法竞赛题目，它们在现实生活中都有着广泛的应用。比如稳定匹配原理作为一种资源的分配方法，就在美国的医疗体系中得到了广泛应用。19 世纪 40 年代，在先进医疗技术的引领下，美国的医疗体系得到了巨大的发展，但是稀缺的医学院毕业生成了这个体系的心病。为了争抢稀缺资源，医院被迫在学生毕业前好几年就向他们提供实习机会。学生们则在还没有被证明有资格从事医疗工作的情况下就已经完成了工作配对，同时，如果医院提供的实习机会没有被学生接受，那么再向别的候选人提供机会就太晚了。很显然，这个市场没有稳定匹配，于是在 19 世纪 50 年代，美国启动了一个名为“国家住院医生匹配项目”（NRMP）的计划，旨在解决这个问题。从 1984 年开始，阿尔文·罗思（Alvin Roth）在论文中研究了这个项目使用的算法并发现了它与 Gale-Shapley 算法原理类似。他随之假设出 NRMP 项目成功的根本原因就是它使用了稳定匹配算法。后来，随着女医生越来越多，情侣们在一个地区寻找实习机会的现象越来越普遍，他们可不喜欢 NRMP 项目的这套匹配机制，这使得情侣们被

很容易安排在两个地方，这又引入了不稳定因素，那就是会导致情侣分居两地。于是在 1995 年，罗思为这个项目设计了一个新算法，这个新算法在 1997 年被 NRMP 所采纳，从那个时候开始到现在，该算法每年为超过 2 万个医生找到了合适的工作岗位。

2012 年诺贝尔经济学奖授予了两位美国学者：阿尔文·罗思和劳埃德·沙普利，以表彰他们在“如何让不同人为了互惠互利而联系在一起”这个课题上的出色研究。没错，这就是本章介绍的 Gale-Shapley 算法中提到的罗思和沙普利，他们被称为“数理经济学家”。

Gale-Shapley 算法又称为“求婚–拒绝算法”（propose-and-reject algorithm），以舞伴问题的整个求解过程来看，女孩从接受第一个邀请开始就有了舞伴，并且舞伴会越来越好，因为女孩可以根据自己的排序表确定是否选择更好的舞伴。与此同时，男孩如果被拒绝，他的选择对象会越来越差（因为男孩是根据自己的排序表从好到差开始选择的）。然而实际情况却并不是这样的，Gale-Shapley 算法中“求婚”的一方总是以最佳可能的稳定岗匹配结束，被“求婚”的一方总是以最差可能的稳定匹配结束，因为选择的主动权掌握在“求婚”者手中。现实生活中的道理也是如此，婚姻中男人如果不主动争取，条件好的女孩就会投入别人的怀抱，留给自己的机会就越来越差。学校里那些勇气可嘉、敢于主动示爱的男生，都是学过 Gale-Shapley 算法的，不信你问问他们。

7.6 参考资料

[1] Gale D, Shapley L S. *College Admissions and the Stability of Marriage*. American Mathematical Monthly，1962,69:9-15

[2] Levitin A. 算法设计与分析基础. 潘彦译. 北京：清华大学出版社，2007

[3] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[4] Knuth D E. *The Art of Computer Programming*(*Third Edition*), Vol 2. Addison-Wesley, 1997

[5] Kleigberg J, Tardos E. *Algorithm Design*. Addison-Wesley, 2005

[6] 稳定配对问题：http://www.docin.com/p-590496944.html

第 8 章 爱因斯坦的思考题

这是一个很有趣的逻辑推理题，传说是爱因斯坦提出来的，他宣称世界上只有 2% 的人能解出这个题目。传说不一定属实，但是这个推理题还是很有意思的。题目是这样的，据说有五个不同颜色的房间排成一排，每个房间里分别住着一个不同国籍的人，每个人都喝一种特定品牌的饮料，抽一种特定品牌的烟，养一种宠物，没有任意两个人抽相同品牌的香烟，或喝相同品牌的饮料，或养相同的宠物。问题是谁在养鱼作为宠物？为了寻找答案，爱因斯坦给出了以下 15 条线索。

1. 英国人住在红色的房子里；
2. 瑞典人养狗作为宠物；
3. 丹麦人喝茶；
4. 绿房子紧挨着白房子，在白房子的左边；
5. 绿房子的主人喝咖啡；
6. 抽 Pall Mall 牌香烟的人养鸟；
7. 黄色房子里的人抽 Dunhill 牌香烟；
8. 住在中间那个房子里的人喝牛奶；
9. 挪威人住在第一个房子里面；
10. 抽 Blends 牌香烟的人和养猫的人相邻；
11. 养马的人和抽 Dunhill 牌香烟的人相邻；
12. 抽 BlueMaster 牌香烟的人喝啤酒；
13. 德国人抽 Prince 牌香烟；
14. 挪威人和住在蓝房子的人相邻；
15. 抽 Blends 牌香烟的人和喝矿泉水的人相邻。

8.1　问题的答案

一般人很难同时记住这么多线索，所以解决这个问题需要用纸和笔画一些表格，一步一步慢慢推理，必要时需要一些假设进行尝试，如果假设错误就推倒重来。我缺乏耐心去做这个事情，所以我一直解不出这个问题。直到有一天，我的一个聪明的朋友告诉我一个答案。我对比了一下前面提到的 15 条线索，发现这是一个正确答案。答案是住在绿色房子里的德国人养鱼作为宠物，完整的推理结果如表 8-1 所示。

我是个懒人，知道了这个问题的答案也就算了，但是我的朋友追问我一个问题，让我不得不正视这个问题。我的朋友想知道这个问题的答案是否唯一，会不会有另外的人推导出另一个完全不同的答案。我想来想去也没有好的办法证明这个问题是否还有其他答案，又懒得自己推理这个问题，只好劳驾任劳任怨的计算机来做这个事情。

表8-1　爱因斯坦思考题推理结果

房子颜色	国　籍	饮　料	宠　物	烟
黄色	挪威	水	猫	Dunhill
蓝色	丹麦	茶	马	Blends
红色	英国	牛奶	鸟	PallMall
绿色	德国	咖啡	鱼	Prince
白色	瑞士	啤酒	狗	BlueMaster

8.2　分析问题的数学模型

整个问题的描述分成两部分，一部分是对问题基本结构的描述，比如每个人住一种颜色的房子，抽一种牌子的香烟，喝一种饮料，等等。另一部分是对线索的描述，比如英国人住在红色的房子中。如果说基本数据结构只是定义了推理结果的一个框架，则线索就可以理解为不同属性之间的绑定关系，用来填充基本结构。因此，对本问题的建模也分成两个部分，一部分是基本模型定义，另一部分是线索模型定义。

8.2.1　基本模型定义

这个问题的描述比较复杂，总结起来共有 5 种颜色的房子、5 种国籍、5 种饮料、5 种宠物和 5 种牌子的香烟，如何用一个数学模型同时表达这 25 个属性呢？这 25 个属性分成 5 种类别，仔细观察这些属性，会发现每个属性都可以用“类型+值”二元组来描述。举个例子，房子颜色是个类型，黄色就是值，组合成“黄色房子”就是一个属性。我们首先将属性的数据结构定义为：

```
typedef struct tagItem
{
    ITEM_TYPE type;
    int value;
}ITEM;
```

ITEM_TYPE 是个枚举类型的量，可以是房子颜色、国籍、饮料类型、宠物类型和香烟牌子五种类型之一，value 是 type 对应的值。value 的取值范围是 0 ~ 4，根据 type 的不同，0 ~ 4 代表的意义也不相同。如果 type 对应的是房子颜色，则 value 取值 0 ~ 4 分别代表蓝色、红色、绿色、黄色和白色，如果 type 对应的是饮料类型，则 value 取值 0 ~ 4 分别代表茶、水、咖啡、啤酒和牛奶。

如果任由这 25 个属性离散存在，会给设计算法带来困难，一般算法建模都会用各种数据结构将这些属性组织起来。观察一下表 8-1 给出的推理结果，我们发现这 25 个属性在两个维度上都存在关系，可以按照类型组织，也可以按照同一推理之间的关系组织，是一个矩阵式关系。根据题目描述，每个人住在一种颜色的房子中，喝一种饮料，养一种宠物，抽一种牌子的香烟，这些关系是固定的，一个人不会同时养两种宠物或喝两种饮料。我们将这种固定的关系称为组（group），一个组中包含一种颜色的房子、一个国籍的人、一种饮料、一种宠物和一种牌子的香烟，他们之间的关系是固定的。既然是这样，可以将 group 数据结构设计为：

```
typedef struct tagGroup
{
    ITEM items[GROUPS_ITEMS];
}GROUP;
```

这样的设计中规中矩，但是会给算法实现带来麻烦，访问每种属性都要遍历 items，通过每个 items 的 type 属性确定要访问的类型。比如要查询或设置房子的颜色，需要遍历 items，找到 items[i].type== type_house 的那个属性进行操作。

在本书中我们多次提到在设计数据结构和算法是利用数组下标的技巧，这里又是一个例子。考虑到上面的麻烦，需要修改 GROUP 的设计，不妨将每种类型在 GROUP 中的位置固定，然后直接利用数据下标进行访问。比如将房子颜色类型固定为数组第一个元素，将国籍固定为数组第二个元素，以此类推，这样 GROUP 定义中可以不需要属性的类型信息（类型信息已经由数组下标表达），只需要一个值信息即可：

```
typedef struct tagGroup
{
    int itemValue[GROUPS_ITEMS];
}GROUP;
```

与此同时，需要对 ITEM_TYPE 枚举类型做值绑定，以便和数组下标对应，绑定值如下：

```
typedef enum tagItemType
{
    type_house = 0,
    type_nation = 1,
    type_drink = 2,
    type_pet = 3,
    type_cigaret = 4
}ITEM_TYPE;
```

使用这种定义数据结构的方式，不仅可以减少设计算法实现的麻烦，还可以提高算法执行效

8

率。比如现在要查看一个 GROUP 绑定组中房子的颜色是否是蓝色，就可以这样编写代码：

```
if(group.itemValue[type_house] == COLOR_BLUE)
```

8.2.2 线索模型定义

接下来考虑一下如何对线索建立数学模型。线索模型的意义在于判断一个枚举结果是否正确，如果某个枚举结果能够符合全部 15 条线索，那这个结果就是最终的正确结果。因此，线索数据结构的定义非常关键，如果定义不好，不仅算法实现会遇到很大的麻烦，而且影响算法实现的效率。即使最后设计出了算法实现，也是到处都是长长的 if…else 分支，本书中多次强调，代码中长长的 if…else 分支结构意味着出现了不良设计。

先分析一下这 15 条线索，大致可以分成三类：第一类是描述某些属性之间具有固定绑定关系的线索，比如，"丹麦人喝茶"和"住绿房子的人喝咖啡"，等等，线索 1、2、3、5、6、7、12、13 可归为此类；第二类是描述某些属性类型所在的"组"所具有的相邻关系的线索，比如，"养马的人和抽 Dunhill 牌香烟的人相邻"和"抽 Blends 牌香烟的人和养猫的人相邻"，等等，线索 10、11、14、15 可归为此类；第三类就是不能描述属性之间固定关系或关系比较弱的线索，比如，"绿房子紧挨着白房子，在白房子的左边"和"住在中间那个房子里的人喝牛奶"，等等。

对于第一类具有绑定关系的线索，其数学模型可以这样定义：

```
typedef struct tagBind
{
    ITEM_TYPE first_type;
    int first_val;
    ITEM_TYPE second_type;
    int second_val;
}BIND;
```

first_type 和 first_val 是一个绑定关系中前一个属性的类型和值，second_type 和 second_val 是绑定关系中后一个属性的类型和值。以线索 6："绿房子的主人喝咖啡"为例，first_type 就是 type_house，first_val 就是 COLOR_GREEN（COLOR_GREEN 是个整数型常量），second_type 就是 type_drink，second_val 就是 DRINK_COFFEE（DRINK_COFFEE 是个整数型常量）。线索 1、2、3、5、6、7、12、13 就可以存储在 binds 数组中：

```
const BIND binds[] =
{
    { type_house, COLOR_RED, type_nation, NATION_ENGLAND },
    { type_nation, NATION_SWEDEND, type_pet, PET_DOG },
    { type_nation, NATION_DANMARK, type_drink, DRINK_TEA },
    { type_house, COLOR_GREEN, type_drink, DRINK_COFFEE },
    { type_cigaret, CIGARET_PALLMALL, type_pet, PET_BIRD },
    { type_house, COLOR_YELLOW, type_cigaret, CIGARET_DUNHILL },
    { type_cigaret, CIGARET_BLUEMASTER, type_drink, DRINK_BEER },
    { type_nation, NATION_GERMANY, type_cigaret, CIGARET_PRINCE }
};
```

对于第二类描述元素所在的“组”具有相邻关系的线索，其数学模型可以这样定义：

```
typedef struct tagRelation
{
    ITEM_TYPE type;
    int val;
    ITEM_TYPE relation_type;
    int relation_val;
}RELATION;
```

type 和 val 是某个“组”内的某个属性的类型和值，relation_type 和 relation_val 是与该属性所在的“组”相邻的“组”中与之有关系的属性的类型和值。以线索 10“抽 Blends 牌香烟的人和养猫的人相邻”为例，type 就是 type_cigaret，val 就是 CIGARET_BLENDS（CIGARET_BLENDS 是个整数型常量），relation_type 是 type_pet，relation_val 是 PET_CAT（PET_CAT 是个整数型常量）。线索 10、11、14、15 就可以存储在 relations 数组中：

```
const RELATION relations[] =
{
    { type_cigaret, CIGARET_BLENDS, type_pet, PET_CAT },
    { type_pet, PET_HORSE, type_cigaret, CIGARET_DUNHILL },
    { type_nation, NATION_NORWAY, type_house, COLOR_BLUE },
    { type_cigaret, CIGARET_BLENDS, type_drink, DRINK_WATER }
};
```

对于第三类线索，无法建立统一的数学模型，只能在枚举算法进行过程中直接使用它们过滤掉一些不符合条件的组合结果。比如线索 8“住在中间那个房子里的人喝牛奶”，就是对每个饮料类型组合结果直接判断 groups[2].itemValue[type_drink]的值是否等于 DRINK_MILK，如果不满足这个线索就不再继续下一个元素类型的枚举。再比如线索 4“绿房子紧挨着白房子，在白房子的左边”，就是在对房子类型进行组合排列时，将绿房子和白房子看成一个整体进行排列组合的枚举，得到的结果直接符合了线索 4 的要求。

8.3 算法设计

和其他穷举类算法一样，本问题的穷举算法也包含两个典型过程，一个是对所有结果的穷举过程，另一个是对结果的证确定判定过程。这两个过程的算法设计与之前的数据结构设计息息相关，本节就分别介绍一下这两个过程的算法设计方法。

8.3.1 穷举所有的组合结果

前面几章也多次介绍穷举法解决问题，但都是一维线性组合的枚举，本题则有些特殊，需要对不同类型的元素分别用穷举法进行枚举，因此不是简单的线性组合。这个算法采用的穷举方法是对不同类型的元素分别进行枚举，然后再按照组的关系组合在一起，这个组合不是线性关系的组合，而是类似阶乘的几何关系的组合。具体思路就是按照 group 中的元素顺序，首先对房子根据颜色组合进行穷举，每得到一组房子颜色组合后，就在此基础上对住在房子里的人的国籍进行

穷举，在房子颜色和国籍的组合结果基础上，在对饮料类型进行穷举，以此类推，直到穷举完最后一种类型得到完整的穷举组合。

这个算法和普通的组合穷举算法不同，需要对五种类型的属性分别枚举，但是每种类型的枚举都有一些特殊情况，比如 8.2.2 节描述的第三类线索。这类情况无法统一处理，需要在枚举算法中进行处理。以枚举房子颜色的算法为例，这里需要处理线索 4“绿房子紧挨着白房子，在白房子的左边”这种特殊情况，请看算法实现：

```
void EnumHouseColors(GROUP *groups, int groupIdx)
{
    if(groupIdx == GROUPS_COUNT) /*递归终止条件*/
    {
        ArrangeHouseNations(groups);
        return;
    }

    for(int i = COLOR_BLUE; i <= COLOR_YELLOW; i++)
    {
        if(!IsGroupItemValueUsed(groups, groupIdx, type_house, i))
        {
            groups[groupIdx].itemValue[type_house] = i;
            if(i == COLOR_GREEN) //应用线索(4)：绿房子紧挨着白房子，在白房子的左边；
            {
                groups[++groupIdx].itemValue[type_house] = COLOR_WHITE;
            }

            EnumHouseColors(groups, groupIdx + 1);
            if(i == COLOR_GREEN)
            {
                groupIdx--;
            }
        }
    }
}
```

这是一个典型的线性枚举，只是在枚举结束的时候继续调用 ArrangeHouseNations()函数继续对房间内住的人的国籍进行枚举。既然绿色房子在白色房子左边，那么每次枚举中只要有绿色房子，就直接将其右边（表现在数据结构中就是下一个组索引）的组中的房子颜色设置成白色。当然，枚举的范围就变成从 COLOR_BLUE 到 COLOR_YELLOW 四种颜色，没有 COLOR_WHITE，因为 COLOR_WHITE 和 COLOR_GREEN 两种颜色直接做了绑定。

对线索 9“挪威人住在第一个房子里面”的特殊处理体现在 ArrangeHouseNations()函数中，请看 ArrangeHouseNations()函数的实现，非常简单吧，这就是数据结构设计带来的便利。

```
void ArrangeHouseNations(GROUP *groups)
{
    /*应用规则(9)：挪威人住在第一个房子里面；*/
    groups[0].itemValue[type_nation] = NATION_NORWAY;
    EnumHouseNations(groups, 1); /*从第二个房子开始*/
}
```

依次完成 5 种属性的枚举，就得到一个类似表 8-1 的完整组合结果，一共有多少种这样的组合结果呢？我们来简单计算一下。首先是对房子颜色进行穷举。因为是 5 种颜色的不重复组合，因此应该有 5! = 120 个颜色组合结果，但是根据线索 4“绿房子紧挨着白房子，在白房子的左边”，相当于绿房子和白房子有稳定的绑定关系，实际就只有 4！ = 24 个颜色组合结果。接下来对 24 个房子颜色组合结果中的每一个结果再进行住户国籍的穷举，理论上国籍也有 5！ = 120 个结果，但是根据线索 9“挪威人住在第一个房子里面”，相当于固定第一个房子住得人始终是挪威人，因此就只有 4！ = 24 个国籍组合结果。穷举完房子颜色和国籍后就已经有 24 × 24 = 576 个组合结果了，接下来需要对这 576 个组合结果中的每一个结果再进行饮料类型的穷举，理论上饮料类型也有 5！ = 120 个结果，但是根据线索 8“住在中间那个房子里的人喝牛奶”，相当于固定了一个饮料类型，因此也只有 4！ = 24 个饮料组合类型。穷举完饮料类型后就得到了 576 × 24 = 13824 个组合结果，接下来对 13824 个组合结果中的每一个结果再进行宠物种类的穷举，这一步没有线索可用，共有 5！ = 120 个结果。穷举完宠物种类后就得到了 13824 × 20 = 1658880 个组合结果，最后对 1658880 个组合结果中的每一个结果再进行香烟品牌的穷举，这一步依然没有线索可用，共有 5！ = 120 个结果。穷举完香烟品牌后就得到了全部组合共 1658880 × 120 = 199065600 个结果。有将近 2 亿个组合结果，看来出现多个正确答案的可能性很大哟。

8.3.2 利用线索判定结果的正确性

根据 8.2.2 节的分析，一共有三类线索，其中第三类线索已经融入到枚举过程中了，因此判断结果的正确性只需要用第一类线索和第二类线索进行过滤即可。第一类线索是同一 `GROUP` 内的属性之间的绑定关系，用来描述的是一个“组”内两种属性之间的固定关系。对这类线索的判断的方法就是遍历全部的“组”，找到 `BIND` 数据中的 `first_type` 和 `first_val` 标识的属性所在的组 `group`，然后检查 `group` 组中类型为 `second_type` 的属性的值是否等于 `second_val`。如果 `group` 中类型为 `second_type` 对应属性的值与 `second_val` 的值不一致就直接返回检查失败，否则就说明当前的组合结果满足此 `BIND` 数据对应的线索，然后对下一个 `BIND` 数据重复上述检查过程，直到检查完 `binds` 数组中所有线索对应的 `BIND` 数据。图 8-1 是用绑定关系线索对结果检查的流程图。

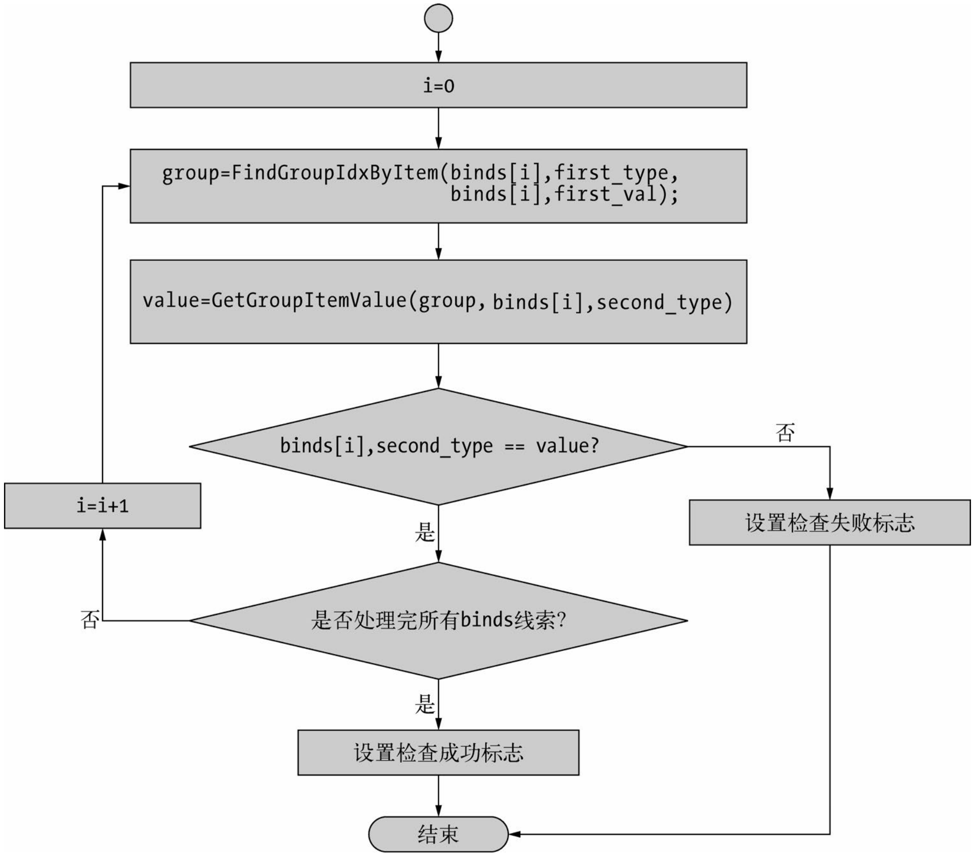


图 8-1　绑定关系线索检查的流程图

第二类线索是“组”之间的相邻关系线索，描述的是相邻的两个组之间的属性的固定关系，判断的方法就是遍历全部的“组”，找到 RELATION 数据中的 type 和 val 标识的元素所在的组 group，然后分别检查与 group 相邻的两个组（第一个组和最后一个组只有一个相邻的组）中类型为 relation_type 的元素对应的值是否等于 relation_val，如果相邻的组中没有一个能满足 RELATION 数据就表示当前组合结果不满足线索，直接返回检查失败。相邻的组中只要一个组中的元素满足 RELATION 数据描述的关系就表示当前组合结果符合 RELATION 数据对应的线索，需要对下一个 RELATION 数据重复上述检查过程，直到检查完 relations 数组中的全部线索对应的 RELATION 数据。图 8-2 是用“组”相邻关系对结果检查的流程图。

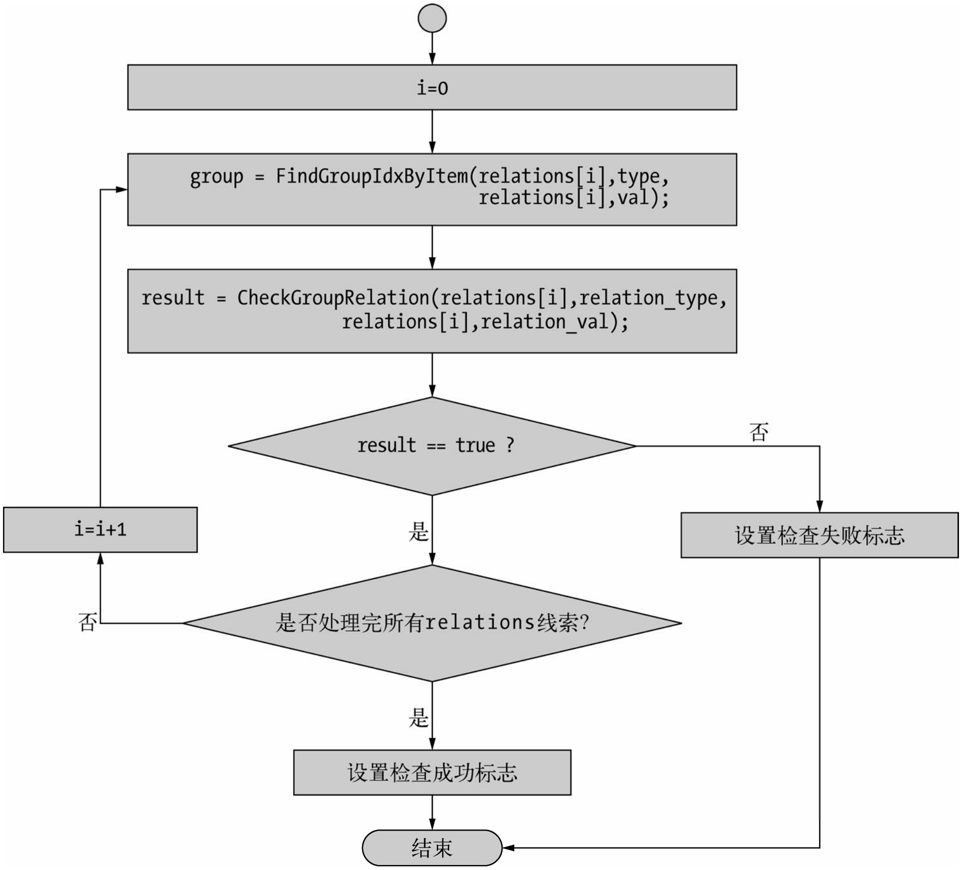


图 8-2 "组"相邻关系线索检查流程图

根据图 8-1 和图 8-2 所示的流程图可以看出，对这两类线索进行检查的算法实现非常简单。得益于我们的数据结构设计，检查算法只需要遍历 binds 数组和 relations 数组即可，避免了写很多 if…else 分支。这两个检查的具体算法实现代码在本书的配书代码中，此处就不再列出。

8.4 总结

虽然有将近 2 亿个组合结果，但是令人惊讶的是，竟然只有一组结果能通过所有的线索检查，就是 8.1 节给出的答案。结果有了，答案真的是唯一的，有点出乎预料，但是也从侧面说明了这个问题的难度。

本问题的穷举算法是一个比较另类的穷举算法，可能因为其结果是二维关系的原因吧，与之前介绍的线性穷举算法稍有不同。通过本算法，大家可以了解一下多个维度穷举的一般方法，就是对每个维度分别穷举，然后再按照关系组合穷举结果。

8.5 参考资料

[1] Levitin A. 算法设计与分析基础. 潘彦译. 北京：清华大学出版社，2007

[2] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[3] Kleigberg J, Tardos E. *Algorithm Design*. Addison-Wesley, 2005

第 9 章 项目管理与图的拓扑排序

作为一个晚睡晚起的典范，我每天早上从睁开眼开始就要在 20 分钟时间内完成以下活动：起床、收听早间新闻、穿衣、洗脸、刷牙、吃早饭、穿鞋、出门赶班车。这些活动一般按照一个合理的顺序依次进行，有些活动也可以同时进行，比如洗脸刷牙的同时可以听新闻，但是我不会一边刷牙一边吃早饭。再比如，穿衣洗漱这些活动一般在出门赶班车之前完成，当然我也可以拿着衣服到班车上穿，但是稍微理智一点我都不会这么做。假如我是一个机器人，谁能告诉我如何在规定的时间内按照合理的顺序完成这些活动？假如每个活动完成都需要一定的时间，比如刷牙需要 3 分钟，洗脸需要 4 分钟，那么如何让我知道能否在 20 分钟内完成这些活动？

起床上班这件事情当然是一件微不足道的小事，说管理就未免太夸大其辞了，但是对于一个由众多活动组成的项目来说，如何对各种关系复杂的活动进行有效的组织，使这些活动能按照合理的顺序逐个完成，就不得不提项目管理。

大家对项目管理都不陌生，无论是大项目还是小项目，最终都可以通过工作分解结构（WBS，Work Breakdown Structure）的方式分解成一系列的任务，然后再细分为具体的活动（activity），每个活动对应一个工作，当这些活动都结束的时候，项目也就完成了。项目中的这些活动都不是孤立的，它们之间存在前后依赖的关系。有的活动没有先决条件，可以安排在任意时间开始，有的活动则依赖其他活动，需要在其依赖的活动都完成后才能开始。面对一堆关系错综复杂的活动，如何安排和组织这些活动，让它们在合适的时间开始，最终能在最短的时间内结束，往往是项目管理者最头疼的事情。幸运的是，有很多项目管理软件帮助人们做这些事情，这些管理软件可以根据开始时间对所有的活动排序，根据这个排序结果就可以知道应该在什么时间安排开始什么活动。项目的执行周期也是管理者最关注的事情之一，管理者需要知道哪些活动最影响项目的时间进度，项目管理软件同样可以找出项目中所有活动的关键路径，盯住关键路径上的活动不延误，项目周期也就有了保证。

举个例子，假如某工程分解后得到 $P_1 \sim P_9$ 共 9 个活动，这些活动之间的依赖关系如表 9-1 所示。

表9-1　工程活动关系表

活动名称	时间（天）	依　赖
P_1	8	
P_2	5	
P_3	6	P_1,P_2
P_4	4	P_3
P_5	7	P_2
P_6	7	P_4,P_5
P_7	4	P_1
P_8	3	P_7
P_9	4	P_4,P_8

将以上活动输入到 Microsoft Office 套件中的 Project 软件中，选择“按照开始时间排序”功能对这些活动进行排序，就可以得到各个活动开始的依次顺序：P_1、P_2、P_5、P_3、P_7、P_8、P_4、P_6、P_9。如果选择“关键路径”功能，软件会提示这个工程的关键路径是：$P_1 \to P_3 \to P_4 \to P_6$，如图 9-1 所示。

对于整个工程项目，人们最担心的是两个问题：一个是工程是否能顺利进行，另一个是估算整个工程完成所需要的最短时间。如果对整个功能的所有活动排序，能得到一个没有环路的活动序列，就说明工程能够顺利进行，同样，只要找到了活动序列中的关键路径，就可以估算出工程完工的最短时间。你有没有想过各种项目管理软件提供的这些功能是如何实现的？其背后又是什么样的算法在支撑这些功能？其实很简单，这些算法用到了图论中的一些理论，对于用图表示的活动序列来说，其对应的操作就是有向图的拓扑排序和关键路径查找，本章就来介绍这些有趣的算法。

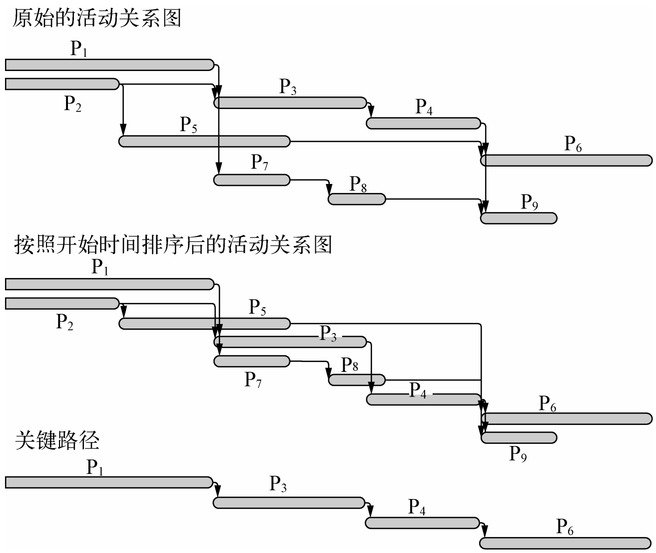


图 9-1　Project 软件的“按照开始时间排序”和“关键路径”功能

9.1 AOV 网和 AOE 网

在图论中，如果某个有向图无法从某个顶点出发经过若干条边回到该点，则称这个图为*有向无环图*（Directed Acyclic Graph，DAG）。有向无环图是描述工程或项目进行过程的有效工具，项目分解后所得到的具体活动之间的关系，可以用有向无环图表示，很显然，这些活动如果存在构成环的顺序依赖关系，会造成环中的活动都无法进行。

图的主要元素是顶点和边，用有向无环图表示工程活动之间的关系时，根据顶点和边所代表的意义不同，通常有两种常见的表示方法，分别是 AOV 网和 AOE 网。如果图中顶点代表的是活动，有向边代表的是与此边相连的两个活动的前后关系，则这样的有向无环图就被称为*顶点表示活动网*（Activity On Vertex network），简称 AOV 网，AOV 网常用于通过拓扑排序决定活动开始关系。图 9-2 就是将表 9-1 中的例子用 AOV 网表示的有向无环图。

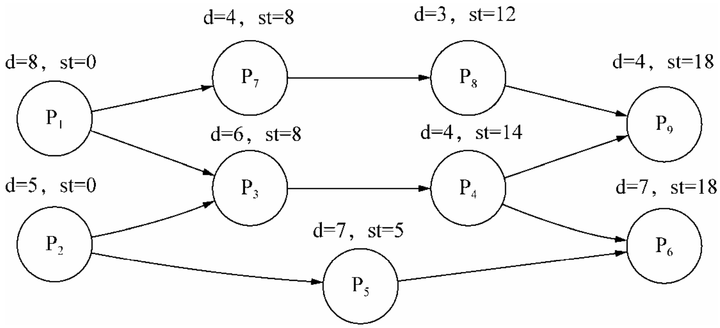


图 9-2　由活动顶点组成的 AOV 网

如果图中边代表的是活动，边的权表示完成活动所需要的时间，与边相连的两个顶点分别表示活动的开始事件和结束事件，则这样的有向无环图就被称为*边表示活动网*（Activity On Edge network），简称 AOE 网，AOE 网是一种带权的有向无环图，AOE 网常用于估算工程完工时间。图 9-3 就是将表 9-1 中的例子用 AOE 网表示的有向无环图。

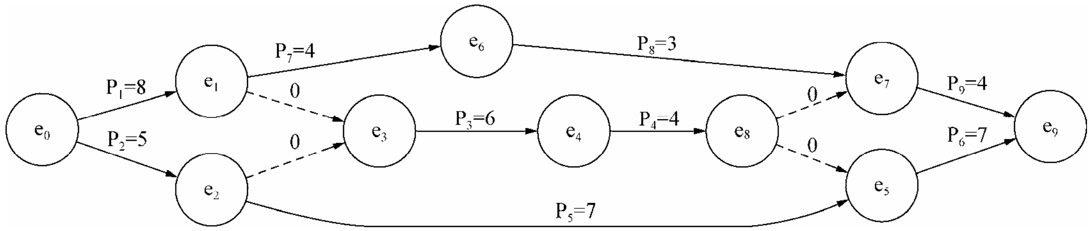


图 9-3　由事件顶点和活动边组成的 AOE 网

9.2　拓扑排序

在图论中，一个有向无环图的所有顶点可以排成一个线性序列，当这个线性序列满足以下条件时，称该序列为一个满足图的*拓扑次序*（topological order）的序列。

- 图中的每个顶点在序列中只出现一次；
- 对于图中任意一条有向边（u, v），在该序列中顶点 u 一定位于顶点 v 之前。

这样的序列也被称为*拓扑序列*，对有向图的所有顶点排序，获得拓扑序列的过程就是有向图的*拓扑排序*（topological sorting）。拓扑排序并不仅仅用于有向图，它是一种利用数据元素中某个属性的偏序关系得到数据元素的全排序序列的方法，本章的内容只关注基于有向图的拓扑排序方法。

9.2.1　拓扑排序的基本过程

对有向图进行拓扑排序可以得到顶点的拓扑序列，拓扑排序的基本过程如下：

(1) 从有向图中选择一个没有前驱（入度为 0）的顶点，输出这个顶点；

(2) 从有向图中删除该顶点，同时删除由该顶点发出的所有有向边。

重复上述步骤(1)和(2)，直到图中不再有入度为 0 的顶点为止。此时，如果所有的顶点都已经输出，则顺序输出的顶点序列就是一个拓扑序列，如果图中还有未输出的顶点，但是入度都不为 0，则说明有向图中存在环路，不能进行拓扑排序。

拓扑排序的现实意义在于，如果按照拓扑序列中的顶点次序安排活动，则在每一项活动开始的时候，能够保证它所依赖的前驱活动都已经完成，从而使得整个工程可以顺序进行，不出现冲突。需要注意的是，对于一个有向无环图来说，有时候不止一个有序的拓扑序列。以表 9-1 所示的工程活动为例，以下三个序列都是合法的拓扑序列：

(1) P_1、P_2、P_5、P_3、P_7、P_8、P_4、P_6、P_9

(2) P_1、P_2、P_7、P_5、P_3、P_8、P_4、P_6、P_9

(3) P_1、P_2、P_5、P_7、P_3、P_8、P_4、P_6、P_9

拓扑排序是根据活动节点进行的，采用 AOV 网的方式展示有向图，可以更直观地看出拓扑序列中各个活动之间的关系。从图 9-2 中可以看出，上述三个拓扑序列的区别仅仅是 P_3、P_5 和 P_7 三个活动的开始次序。

9.2.2　按照活动开始时间排序

工程实施过程中，人们总是希望每个活动尽早开始。对一个工程的所有活动进行拓扑排序时，如果能将活动的最早开始时间考虑进来，让拓扑序列中的每个活动都尽早开始，这样的排序序列对工程实施具有非常大的实用性。Project 软件中的“按照开始时间排序”就是满足这种需求的一

个功能。这一节我们也仿照 Project 软件实现一个按照开始时间对活动进行拓扑排序的算法。当然，这背后其实就是拓扑排序，大家可以体会一下算法在实际生活中的应用。

在一个工程中，每个活动的开始时间受前置活动的约束，不可能随时开始。但是活动的开始时间可以根据前置活动之间的关系推算出来，具体推算的方法如下。

- 如果一个活动没有前驱活动，则这个活动的开始时间是 0；
- 如果一个活动有前驱活动，则这个活动的开始时间是前驱活动的开始时间和前驱活动持续时间的和，如果一个活动有多个前驱活动，则这个活动的开始时间是这些和中最大的一个。

以表 9-1 中的活动为例，P_1 和 P_2 没有前驱活动，其开始时间 st=0，P_7 的开始时间是 P_1 的开始时间和 P_1 的持续时间之和，即 P_7 的开始时间 st=8。P_3 的开始受制于 P_1 和 P_2，其开始时间是 max(8+0, 5+0)，即 P_3 的开始时间 st=8。最终每个活动的开始时间如图 9-2 所示，现在我们就基于这个开始时间的对表 9-1 中的活动进行拓扑排序。

对于 AOV 网，用邻接表方式定义有向图的数据是最常用的方式，对于图 9-2 所表示的有向图，用邻接表方式定义的数据结构应该如图 9-4 所示。首先定义图的顶点，其数据结构描述如下所示。

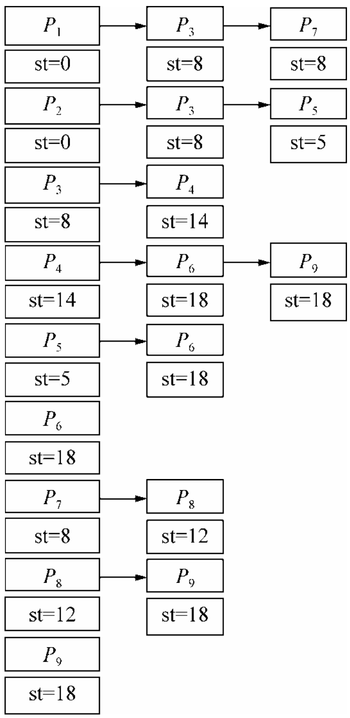


图 9-4　图的邻接表表达形式

```
typedef struct tagVertexNode
{
    char *name;   //活动名称
    int days;     //完成活动所需时间
    int sTime;    //活动最早开始时间
    int inCount;  //活动的前驱节点个数
    int adjacent; //相邻活动的个数
    int adjacentNode[MAX_VERTEXNODE]; //相邻活动列表(节点索引)
}VERTEX_NODE;
```

对于 adjacentNode 属性需要特别说明一下，这个列表中存储的是邻接顶点在图中的节点索引，如果图采用数组方式组织所有的顶点，则这个表中存储的就是邻接顶点在数组中的位置（数组下标）。图的定义如下：

```
typedef struct tagGraph
{
    int count; //图的顶点个数
    VERTEX_NODE
vertexs[MAX_VERTEXNODE]; //图的顶点列表
}GRAPH;
```

根据 9.2.1 节介绍的基本拓扑排序过程，有两个细节需要算法特殊处理，其一是同一时刻有多个入度为 0 的顶点的情况如何处理，其二是删除一个顶点发出的所有有向边后对新产生的入度为 0 的顶点如何处理。基本排序过程对这些情况不做任何特殊处理，也就是说对于同时出现的入度为 0 的顶点，以任何次序输出都是合法的。但是如果考虑开始时间属性，就需要对这些入度为 0 的顶点按照开始时间排序，才能保证最后输出是按照开始时间拓扑排序的结果。

按照基本拓扑排序要求，同时出现的入度为 0 的顶点要按照“先进先出”的原则进行处理，同时，还要能够根据开始时间排排序。针对这种情况，使用“优先级队列”管理算法处理过程中同时出现的入度为 0 的顶点就是一个最好的选择。这些顶点首先按照出现的先后次序入队，同时根据开始时间调整在队列中的位置，保证开始时间较小的顶点能先于开始时间较大的顶点输出。

正如 TopologicalSorting() 函数代码所展示的那样，整个排序算法的核心就是对这个“优先级队列”的处理。算法的第一步就是遍历有向图的所有顶点，将所有入度为 0（inCount 值为 0）的顶点入队，这一步完成以后优先级队列中的两个元素是 P_1 和 P_2 两个顶点。算法的第二部分就是围绕这个优先级队列进行处理，首先出队的是 P_1，同时删除 P_1 发出的两条有向边，删除有向边的操作是通过减少与之相邻的顶点的 inCount 值来实现的，删除 P_1 的有向边会导致 P_7 顶点的入度为 0，因此 P_7 加入队列，因为 P_7 的开始时间是 8，因此 P_7 排在 P_2 之后，此时队列中剩下的两个元素分别是 P_2 和 P_7。第二轮队列处理时，P_2 出队，同时删除 P_2 发出的两条有向边，这会导致 P_3 和 P_5 两个顶点的入度为 0，P_3 和 P_5 分别入队，但是 P_5 的开始时间是 5，小于 P_3 和 P_7 的 8，P_5 排在队列的最前面，此时队列中的元素分别是 P_5，P_3 和 P_7。重复以上过程，直到队列为空时算法结束，此时判断输出的排序列表 sortedNode，如果 sortedNode 中的节点个数与图的顶点个数相同，则说明所有顶点都已经输出，拓扑排序完成，否则就说明图中存在环路，无法进行拓扑排序。拓扑排序的代码实现如下所示。

```
bool TopologicalSorting(GRAPH *g, std::vector<int>& sortedNode)
{
    std::priority_queue<QUEUE_ITEM> nodeQueue;

    for(int i = 0; i < g->count; i++)
    {
        if(g->vertexs[i].inCount == 0)
        {
            EnQueue(nodeQueue, i, g->vertexs[i].sTime);
        }
    }

    while(nodeQueue.size() != 0)
    {
        int node = DeQueue(nodeQueue); //按照开始时间优先级出队
        sortedNode.push_back(node);//输出当前节点
        //遍历节点 node 的所有邻接点，将表示有向边的 inCount 值减 1
        for(int j = 0; j < g->vertexs[node].adjacent; j++)
        {
            int adjNode = g->vertexs[node].adjacentNode[j];
            g->vertexs[adjNode].inCount--;
            //如果 inCount 值为 0，则该节点入队列
            if(g->vertexs[adjNode].inCount == 0)
            {
                EnQueue(nodeQueue, adjNode, g->vertexs[adjNode].sTime);
            }
        }
    }

    return (sortedNode.size() == g->count);
}
```

根据表 9-1 的活动数据构造有向图，然后调用 TopologicalSorting()函数得到按照时间排序的活动拓扑序列，与 Project 软件输出的排序结果一致。需要说明一点，使用 TopologicalSorting()函数之前需要手工计算好每个节点的最早开始时间，最早开始时间的自动计算算法，将在 9.3 节介绍。

9.3 关键路径算法

前面提到过，对于工程管理，人们最关注的两个问题分别是工程是否能顺利进行，以及估算整个工程完成所需要的最短时间和影响工程时间的关键活动。前一个问题可用拓扑排序解决，后一个问题则需要找出工程进行的关键路径，关键路径上的活动完成所需要的时间就是工程完成所需要的最短时间。关键路径通常是所有工程活动中最长的路径，关键路径上的活动如果延期将直接导致工程延期。

利用 AOV 网表示有向图，可以对活动进行拓扑排序，根据排序结果对工程中活动的先后顺序做出安排。但是寻找关键路径，估算工程活动的结束时间，则需要使用 AOE 网表示有向图。AOE 网中用顶点表示事件，有向边表示活动，边上的权值表示活动持续的时间。只有在某顶点

所代表的事件发生后，从该顶点出发的各有向边所代表的活动才能开始，反之亦然，只有在指向某一顶点的各有向边所代表的活动都已经结束后，该顶点所代表的事件才能发生。AOE 网只有一个入度为 0 的顶点（源点）和一个出度为 0 的顶点（汇点），分别代表开始事件和结束事件，其他的顶点则表示两个意义，其一是此点之前的所有活动都已经结束，其二是此点之后的活动可以开始了。对于表 9-1 所列举的活动，用 AOE 网表示的结果如图 9-3 所示，其中虚线连接的顶点表示两个事件是同质事件，也就是说这两个顶点代表相同的事件，边的权是 0 表示这两个顶点之间没有活动。

计算关键路径的算法需要根据 AOE 网的特征调整图的数据结构定义，本节介绍的算法仍然使用邻接表来表示图，但是需要重新定义顶点和边的数据结构。因为 AOE 网的边代表具体的活动，需要在数据结构中明确体现“边”的定义，调整后的边和顶点的定义如下所示：

```
typedef struct tagEdgeNode
{
    int vertexIndex;    //活动边终点顶点索引
    std::string name;   //活动边的名称
    int duty;           //活动边的时间（权重）
}EDGE_NODE;

typedef struct tagVertexNode
{
    int sTime;    //事件最早开始时间
    int eTime;    //事件最晚开始时间
    int inCount;  //活动的前驱节点个数
    std::vector<EDGE_NODE> edges; //相邻边表
}VERTEX_NODE;
```

算法开始之前，每个顶点的 `sTime` 被初始化为 0，`eTime` 被初始化为一个有效范围之外的最大值（0x7FFFFFFF），算法结束之后，`sTime` 和 `eTime` 会被计算为实际的时间值。

9.3.1 什么是关键路径

开始讨论关键路径之前，先来介绍一下活动的最早开始时间和最晚开始时间。工程中一个活动何时开始依赖于其前驱活动何时结束，只有所有的前驱活动都结束后这个活动才可以开始，前驱活动都结束的时间就是这个活动的最早开始时间。与此同时，在不影响工程完工时间的前提下，有些活动的开始时间存在一些余量，在时间余量允许的范围之内推迟一段时间开始活动也不会影响工程的最终完成时间，活动的最早开始时间加上这个时间余量就是活动的最晚开始时间。活动不能在最早开始时间之前开始，当然，也不能在最晚开始时间之后开始，否则会导致工期延误。

如果一个活动的时间余量为 0，即该活动的最早开始时间和最晚开始时间相同，则这个活动就是关键活动，由这些关键活动串起来的一个工程活动路径就是关键路径。根据关键路径的定义，一个工程中的关键路径可能不止一个，我们常说的关键路径指的是工程时间最长的那条路径，也就是从源点到汇点之间最长的那条活动路径。

9.3.2 计算关键路径的算法

根据 9.3.1 节的介绍，计算关键路径的基础是先找出工程中的所有关键活动，确定一个活动是否是关键活动的依据就是活动的最早开始时间和最晚开始时间，因此需要先介绍如何计算活动的最早开始时间和最晚开始时间。在 AOE 网中，事件 e_i 必须在指向 e_i 的所有活动都结束后才能发生，只有 e_i 发生之后，从 e_i 发出的活动才能开始，因此 e_i 的最早发生时间就是 e_i 发出的所有活动的最早开始时间。如果用 est[i]表示事件 e_i 的最早开始时间，用 duty[i,j]表示连接事件 e_i 和事件 e_j 的活动需要持续的时间，则事件 e_i 的最早开始时间可以用以下关系推算：

(1) est[0] = 0

(2) est[n] = max{est[i]+duty[i,n], est[j]+duty[j,n], ⋯, est[k]+duty[k,n]}

（其中 $I,j,\cdots,k$ 是事件 n 的前驱事件）

根据以上推算关系，可以将图 9-3 中的 $e_0 \sim e_3$ 几个事件的最早开始时间推算出来：

est[0] = 0

est[1] = est[0]+duty[0,1] = 0+8 = 8

est[2] = est[0]+duty[0,2] = 0+5 = 5

est[3] = max{est[1]+duty[1,3], est[2]+suty[2,3]} = max{8+0, 5+0} = 8

很显然，这个推算关系是建立在合法的拓扑序列的基础上的，因此，推算事件的最早开始时间需要对图中的事件节点进行拓扑排序。拓扑排序的算法已经在 9.2 节介绍过了，现在我们只关注最早开始时间的计算方法。假设 `sortedNode` 参数中存放的图的拓扑排序结果，`CalcESTime()`函数从拓扑序列的第一个顶点开始（变量 `u` 代表的顶点），遍历这个顶点发出的有向边指向的相邻顶点（变量 `v` 代表的顶点），如果该顶点的最早开始时间与有向边代表的活动持续时间的和（这个结果存放在临时变量 `uvst` 中）大于有向边指向的相邻顶点的最早开始时间，则更新这个相邻顶点的最早开始时间。需要注意的是，算法并没有直接利用推算关系中的 max 选择处理，而是按照 `sortedNode` 序列中的顶点先后关系，只在处理到相邻顶点时才更新最早开始时间（这正是所有顶点的 `sTime` 被初始化成 0 的原因），当 `sortedNode` 序列中的所有顶点都处理完之后，就相当于变相地实现了 max 选择的处理。

```
void CalcESTime(GRAPH *g, const std::vector<int>& sortedNode)
{
    g->vertexs[0].sTime = 0; //est[0] = 0

    std::vector<int>::const_iterator nit = sortedNode.begin();
    for(; nit != sortedNode.end(); ++nit)
    {
        int u = *nit;
        //遍历 u 出发的所有有向边
        std::vector<EDGE_NODE>::iterator eit = g->vertexs[u].edges.begin();
        for(; eit != g->vertexs[u].edges.end(); ++eit)
        {
            int v = eit->vertexIndex;
```

9

```
                int uvst = g->vertexs[u].sTime + eit->duty;
                if(uvst > g->vertexs[v].sTime)
                {
                    g->vertexs[v].sTime = uvst;
                }
            }
        }
    }
```

事件 e_i 的最晚开始时间定义为：e_i 的后继事件 e_j 的最晚开始时间减去 e_i 和 e_j 之间的活动的持续时间的差，当 e_i 有多个后继事件时，则取这些差值中最小的一个作为 e_i 的最晚开始时间。如果用 lst[j]表示事件 e_j 的最晚开始时间，用 duty[i, j]表示事件 e_i 和后继事件 e_j 之间的活动需要持续的时间，则事件 e_i 的最晚开始时间可以用以下关系推算：

(1) lst[n] = est[n]

(2) est[i] = min{lst[j]–duty[i,j], est[k]–duty[i,k], ⋯, est[m]–duty[i,m]}

（其中 j,k,⋯,m 是事件 i 的后继事件）

仍然以图 9-3 为例，我们推算一下 e_5、e_7、e_8 和 e_9 几个事件的最晚开始时间：

lst[9] = est[9] = 25

lst[5] = lst[9]–duty[5,9] = 25–7 = 18

lst[7] = lst[9]–duty[7,9] = 25–4 = 21

lst[8] = min{lst[7]–duty[8,7], lst[5]–duty[8,5]} = min{21–0, 18–0} = 18

这个最晚开始时间的推算关系是建立在合法的拓扑序列的逆序基础上的，CalcLSTime()函数对 sortedNode 序列的处理顺序和 CalcESTime()函数刚好相反，从拓扑序列的最后一个顶点（变量 u 代表的顶点）开始向前遍历。如果该顶点的后继顶点（变量 v 代表的顶点）的最晚开始时间与连接这两个顶点的活动的持续时间的差小于该顶点（u 顶点）的最晚开始时间，则更新该顶点的最晚开始时间。和 CalcESTime()函数一样，CalcLSTime()函数也没有直接利用 min 选择处理，但是通过逆序遍历 sortedNode 序列中的所有顶点，变相地实现了 min 选择的处理。

```
void CalcLSTime(GRAPH *g, const std::vector<int>& sortedNode)
{
    //最后一个节点的最晚开始时间等于最早开始时间
    g->vertexs[g->count - 1].eTime = g->vertexs[g->count - 1].sTime;

    std::vector<int>::const_reverse_iterator cit = sortedNode.rbegin();
    for(; cit != sortedNode.rend(); ++cit)
    {
        int u = *cit;
        //遍历 u 出发的所有有向边
        std::vector<EDGE_NODE>::iterator eit = g->vertexs[u].edges.begin();
        for(; eit != g->vertexs[u].edges.end(); ++eit)
        {
            int v = eit->vertexIndex;
            int uvet = g->vertexs[v].eTime - eit->duty;
            if(uvet < g->vertexs[u].eTime)
```

```
                {
                    g->vertexs[u].eTime = uvet;
                }
            }
        }
    }
```

在 AOE 网中计算好每个顶点代表的事件的最早开始时间和最晚开始时间之后，就可以很容易计算出每条边代表的活动的最早开始时间和最晚开始时间。假如某个活动两端的事件分别是 e_i 和 e_j，则该活动的最早开始时间就是事件 e_i 的最早开始时间，该活动的最晚开始时间就是事件 e_j 的最晚开始时间减去该活动的持续时间。用这个关系计算出所有活动的最早开始时间和最晚开始时间，只要最早开始时间和最晚开始时间相同的活动都是关键活动，按照事件顶点的拓扑序列的先后关系，顺序输出这些事件顶点相关的关键活动，得到的关键活动序列就是关键路径。

综合前面的分析，计算关键路径的需要以下四个步骤。

(1) 对事件顶点进行拓扑排序，得到事件的拓扑序列；

(2) 计算事件顶点的最早开始时间；

(3) 计算事件顶点的最晚开始时间；

(4) 计算活动的最早开始时间和最晚开始时间，并按照事件的拓扑顺序逐次输出关键活动，得到关键路径。

这四个步骤非常清晰地体现在 CriticalPath()函数中，重点是第四个步骤输出关键路径。判断活动是否是关键活动是通过这行 if 语句实现的：

```
if(g->vertexs[u].sTime == g->vertexs[v].eTime - eit->duty)
```

但是要实现按照活动顺序输出关键活动路径的功能，还需要按照事件顶点拓扑排序的结果逐个判断每个事件发出的活动（就是事件顶点发出的有向边），按照活动的开始次序逐个输出关键活动。CriticalPath()函数中的第一个 for 循环就是按照拓扑排序的结果逐个处理事件顶点，第二个 for 循环就是搜索一个顶点的所有有向边，查找关键活动。需要注意的是，图 9-3 中虚线画出的边是实际不存在的虚拟活动，虽然不影响结果，但是也会被当成关键活动输出，因此需要判断一下，如果是虚拟活动则需要过滤一下。CriticalPath()函数在输出关键路径时没有做过滤处理，过滤的方法其实也很简单，根据 name 是否为空或活动时间是否是 0 都可以作为判断过滤的依据，有兴趣的读者可自行完成。

```
bool CriticalPath(GRAPH *g)
{
    std::vector<int> sortedNode;
    if(!TopologicalSorting(g, sortedNode)) //步骤 1
    {
        return false;
    }
    CalcESTime(g, sortedNode); //步骤 2
    CalcLSTime(g, sortedNode); //步骤 3
    //步骤 4：输出关键路径上的活动名称
    std::vector<int>::iterator nit = sortedNode.begin();
```

```
    for(; nit != sortedNode.end(); ++nit)
    {
        int u = *nit;
        std::vector<EDGE_NODE>::iterator eit = g->vertexs[u].edges.begin();
        for(; eit != g->vertexs[u].edges.end(); ++eit)
        {
            int v = eit->vertexIndex;
            if(g->vertexs[u].sTime == g->vertexs[v].eTime - eit->duty)
            {
                std::cout << eit->name << std::endl;
            }
        }
    }

    return true;
}
```

对于表 9-1 的活动关系数据，转化成 AOE 网形式的有向图之后，用 `CriticalPath()`函数计算出的关键路径是 $P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P_6$，与 Project 软件计算出的关键路径结果是一样的。我们用自己写的算法实现了一样的功能，这些软件也是使用相同的算法，并无太多神秘可言。

9.4　总结

现在回到本章章首的那个例子，一组没有任何关系的活动，在一定的规则或常识的约束下，在活动的某个属性（开始时间）上形成了或弱或强的顺序关系，这就是一个偏序，在这个偏序上排序得到的一个全序就是这组活动的拓扑排序。这种偏序关系的强弱取决于规则和约束力的大小，正常情况下穿衣服应该在坐班车之前发生，但是我也可以选择在班车上穿衣服。当然，这取决于我失去理智的程度。

生活中有很多看似神奇但是原理却很简单的东西，本章介绍的两个算法就是这样，工程管理软件中最实用的两个功能，原来就是两个简单的算法在背后支撑其实现。很多软件，无论是动辄几百兆字节的大型软件，还是几十千字节的小程序，背后都是不同的算法在支撑其展示的各种功能，没有任何神秘可言，本章给出的例子只是这类软件功能的冰山之一角罢了。

这两个算法用到了图、数组和带优先级标记的队列等数据结构，灵活地运用这些数据结构给算法实现带来了极大的便利，比如，普通的拓扑排序和本章给出的按照开始时间排序在算法结构上完全一样，唯一的区别就是使用带优先级标记的队列代替普通的队列。

9.5　参考资料

[1]　维基百科：http://zh.wikipedia.org/wiki/拓扑排序

[2]　Cormen T H , et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[3]　维基百科：http://zh.wikipedia.org/wiki/关键路径

第 10 章 RLE 压缩算法与 PCX 图像文件格式

RLE（Run Length Encoding）压缩算法即行程长度压缩算法，也称游程长度压缩算法，是最早出现、也是最简单的无损数据压缩算法。RLE 压缩算法对于黑白图像和基于调色板的单调图像有很高的压缩效率，不仅常用于处理图像数据，在传真机上也得到了广泛的应用。

10.1 RLE 压缩算法

RLE 压缩算法（下简称 RLE 算法）的基本思路是把数据按照线性序列分成两种情况：一种是连续的重复数据块，另一种是连续的不重复数据块。RLE 算法的原理就是用一个表示块数的属性加上一个数据块代表原来连续的若干块数据，从而达到节省存储空间的目的。一般 RLE 算法都选择数据块的长度为 1 字节，表示块数的属性也用 1 字节表示，对于颜色数小于 256 色的图像文件或文本文件，块长度选择 1 字节是比较合适的。

10.1.1 连续重复数据的处理

RLE 算法有很多优化和改进的变种算法，这些算法对连续重复数据的处理方式基本上都是一样的。对于连续重复出现的数据，RLE 算法一般用两字节表示原来连续的多字节重复数据。我们用一个例子更直观地说明 RLE 算法对这种情况的处理，假如原始数据有 5 字节的连续数据：

```
[data] [data] [data] [data] [data]
```

则压缩后的数据就包含块数和[data]两字节，其中[data]只存储了一次，节省了存储空间：

```
[5] [data]
```

需要注意的是，一般 RLE 算法都采用插入一个长度属性字节存储连续数据的重复次数，因此能够表达的最大值就是 255 字节，如果连续的相同数据超过 255 字节时，就从第 255 字节处断开，将第 256 字节以及 256 字节后面的数据当成新的数据处理。随着 RLE 算法采用的优化方式不同，这个长度属性字节所表达的意义也不同，对于本章给出的这种优化算法，长度属性字节的最高位被用来做一个标志位，只有 7 位用来表示长度，这一点在下一节会具体说明。

10.1.2 连续非重复数据的处理

对于连续的非重复数据，RLE 算法有两种处理方法，一种处理方法是将每个不重复的数据当作只重复一次的连续重复数据处理，在算法实现上就和处理连续重复数据一样；另一种处理方法是不对数据进行任何处理，直接将原始数据作为压缩后的数据存储。假如有以下 5 字节的连续非重复数据：

```
[data1] [data2] [data3] [data4] [data5]
```

按照第一种处理方法，最后的压缩数据就如下所示：

```
[1][data1] [1][data2] [1][data3] [1][data4] [1][data5]
```

如果按照第二种处理方法，最后的数据和原始数据一样：

```
[data1] [data2] [data3] [data4] [data5]
```

如果采用第一种方式处理连续非重复数据，则存在一个致命的问题，对连续出现的不重复数据，会因为插入太多块数属性字节而膨胀一倍，如果原始数据主要是随机的非重复数据，则采用这种方式不仅不能起到压缩数据的目的，反而起到恶化的作用。多数经过优化的 RLE 算法都会选择使用第二种方式处理连续非重复数据，但是这就引入了新问题，在 RLE 算法解码的时候，如何区分连续重复和非重复数据?

前面已经提到，如果把非重复数据当作独立的单次重复数据处理，反而会造成数据膨胀，但是如果把连续非重复数据也当成一组数据整理考虑呢？这是一个优化的思路，首先，给连续重复数据和连续非重复数据都附加一个表示长度的属性字节，并利用这个长度属性字节的最高位来区分两种情况。长度属性字节的最高位如果是 1，则表示后面紧跟的是个重复数据，需要重复的次数由长度属性字节的低 7 位（最大值是 127）表示。长度属性字节的最高位如果是 0，则表示后面紧跟的是非重复数据，长度也由长度属性字节的低 7 位表示。

采用这种优化方式，压缩后的数据非常有规律，两种类型的数据都从长度属性字节开始，除了标志位的不同，后跟的数据也不同。第一种情况后跟一个字节的重复数据，第二种情况后跟的是若干个字节的连续非重复数据。

10.1.3 算法实现

首先介绍一下数据压缩的编码过程如何实现。采用 10.1.2 节给出的优化方式，编码算法不仅要能够识别连续重复数据和连续非重复数据两种情况，还要能够统计出两种情况下数据块的长度。编码算法从原始数据的起始位置开始向后搜索，如果发现后面是重复数据且重复次数超过 2，则设置连续重复数据的标志并继续向后查找，直到找到第一个与之不相同的数据为止，将这个位置记为下次搜索的起始位置，根据位置差计算重复次数，最后长度属性字节以及一个字节的原始重复数据一起写入压缩数据；如果后面数据不是连续重复数据，则继续向后搜索查找连续重复数据，直到发现连续重复的数据且重复次数大于 2 为止，然后设置不重复数据标志，将新位置记为

下次搜索的起始位置，最后将长度属性字节写入压缩数据并将原始数据逐字节复制到压缩数据。然后从上一步标记的新的搜索起始位开始，一直重复上面的过程，直到原始数据结束。

```
int Rle_Encode(unsigned char *inbuf, int inSize, unsigned char *outbuf, int onuBufSize)
{
    unsigned char *src = inbuf;
    int i;
    int encSize = 0;
    int srcLeft = inSize;

    while(srcLeft > 0)
    {
        int count = 0;
        if(IsRepetitionStart(src, srcLeft)) /*是否连续三个字节数据相同? */
        {
            if((encSize + 2) > onuBufSize) /*输出缓冲区空间不够了*/
            {
                return -1;
            }
            count = GetRepetitionCount(src, srcLeft);
            outbuf[encSize++] = count | 0x80;
            outbuf[encSize++] = *src;
            src += count;
            srcLeft -= count;
        }
        else
        {
            count = GetNonRepetitionCount(src, srcLeft);
            if((encSize + count + 1) > onuBufSize) /*输出缓冲区空间不够了*/
            {
                return -1;
            }
            outbuf[encSize++] = count;
            for(i = 0; i < count; i++) /*逐个复制这些数据*/
            {
                outbuf[encSize++] = *src++;;
            }
            srcLeft -= count;
        }
    }
    return encSize;
}
```

Rle_Encode()函数是 RLE 算法的实现，它通过调用 IsRepetitionStart()函数判断从 src 开始的数据是否是连续重复数据，如果是连续重复数据，则调用 GetRepetitionCount()函数计算出连续重复数据的长度，将长度属性字节的最高位置 1 并向输出缓冲区写入一个字节的重复数据。如果不是连续重复数据，则调用 GetNonRepetitionCount()函数计算连续非重复数据的长度，将长度属性字节的最高位置 0 并向输出缓冲区复制连续的多个非重复数据。GetRepetitionCount()函数和 GetNonRepetitionCount()函数都比较简单，此处就不列出代码了，你可以在本章的随书代码中找到它们。根据算法要求，只有数据重复出现两次以上才算作连续重复数据，因此

IsRepetitionStart()函数检查连续的 3 字节是否是相同的数据，如果是则判定为出现连续重复数据。之所以要求至少要 3 字节的重复数据才判定为连续重复数据，是为了尽量优化对短重复数据间隔出现时的压缩效率。举个例子，对于这样的数据“AABCCD”，如果不采用这个策略，最终的压缩数据应该是[0x82][A][0x01][B][0x82][C][0x01][D]，压缩后数据长度是 8 字节。如果采用这个策略，则上述数据就被认定为连续非重复数据局，最终被压缩为[0x06][A][A][B][C][C][D]，压缩后数据长度是 7 字节，这样的数据越长，效果越明显。

解压缩算法相对比较简单，因为两种情况下的压缩数据首部都是 1 字节的长度属性标识，只要根据这个标识判断如何处理就可以了。首先从压缩数据中取出 1 字节的长度属性标识，然后判断是连续重复数据的标识还是连续非重复数据的标识，如果是连续重复数据，则将标识字节后面的数据重复复制 *n* 份写入输出缓冲区；如果是连续非重复数据，则将标识字节后面的 *n* 个数据复制到输出缓冲区。*n* 的值是标识字节与 0x7F 做与操作后得到，因为标识字节低 7 位就是数据长度属性。

```
int Rle_Decode(unsigned char *inbuf, int inSize, unsigned char *outbuf, int onuBufSize)
{
    unsigned char *src = inbuf;
    int i;
    int decSize = 0;
    int count = 0;

    while(src < (inbuf + inSize))
    {
        unsigned char sign = *src++;
        int count = sign & 0x7F;
        if((decSize + count) > onuBufSize) /*输出缓冲区空间不够了*/
        {
            return -1;
        }
        if((sign & 0x80) == 0x80) /*连续重复数据标志*/
        {
            for(i = 0; i < count; i++)
            {
                outbuf[decSize++] = *src;
            }
            src++;
        }
        else
        {
            for(i = 0; i < count; i++)
            {
                outbuf[decSize++] = *src++;
            }
        }
    }

    return decSize;
}
```

Rle_Decode()函数是解压缩算法的实现代码，每组数据的第一字节是长度标识字节，其最高位是标识位，低 7 位是数据长度属性，根据标识位分别进行处理即可。

10.2 RLE 与 PCX 图像文件格式

PCX 图像文件格式是 Zsoft 公司在 20 世纪 80 年代初期设计的一种图像文件格式，专用于存储该公司开发的 PC Paintbrush 绘图软件所生成的图像画面数据。在计算机普遍只能处理单色图像的年代，PCX 图像文件格式就超前地引入了彩色图像的概念，使得 PCX 文件一度成为 DOS 时代 PC 机上最流行的图像标准之一。PCX 图像文件格式不仅支持单色图像文件，还支持 16 色和 256 色的彩色图像文件。PCX 文件采用 RLE 算法，对存储绘图类型的图像（大块连续色调的图像）具有比较高的压缩效率，但是对于照片和视频图像效果不佳。

其中 256 色图像文件的格式非常简单，我们就以 256 色 PCX 文件为例，介绍一下 RLE 算法在 PCX 文件中的应用。

10.2.1 PCX 图像文件格式

PCX 文件可以分成三类：单色 PCX 文件（黑白图像）、少于 16 种颜色的彩色 PCX 文件、256 种颜色的 PCX 文件。最初的 PCX 图像文件格式被设计为一种与特定图形显示硬件密切相关的图像处理格式，比如少于 16 种颜色的 PCX 文件，其数据被分成 4 个颜色层存放，这与 EGA 和 VGA 处理 16 色的方式密切相关（4 个颜色位平面），256 色的 PCX 文件与 SVGA 处理彩色图像的方式有关。PCX 文件图像结构如图 10-1 所示。

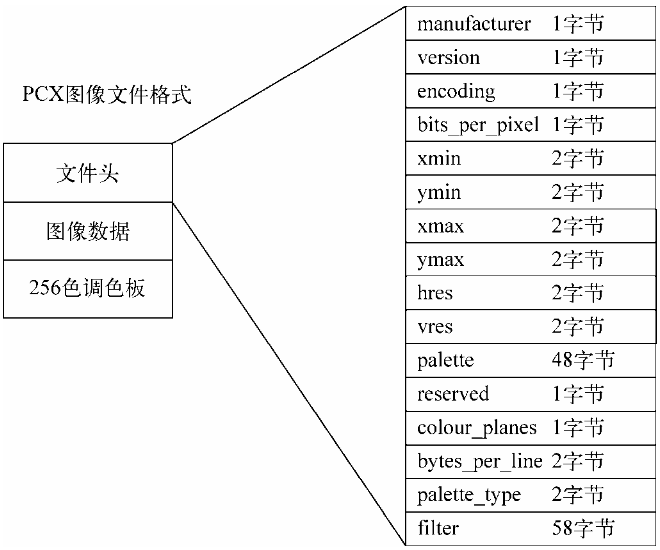


图 10-1 PCX 文件结构示意图

PCX 文件结构分三个域，分别是文件头、图像数据和最后附加的 256 色调色板数据。文件头是一个大小和格式都固定的表头，长度是 128 字节，主要存放定义图像的尺寸，彩色调色板及其他有关的图像数据。文件头中的 manufacture 字节是固定值 0x0a，这是识别 PCX 文件的唯一标志。version 字节说明 PCX 文件的版本，其中 5 对应的版本开始支持 256 色图像。encoding 字节表示图像数据压缩编码方式，其值为 1 时表示采用 RLE 压缩编码的方法。bits_per_pixel 说明每个像素的位数，对于 256 色 PCX 图像，这个值要么不用，要么是 8。xmin、ymin、xmax 和 ymax 四个值定义文件的图像尺寸，xmax - xmin 的值是图像的实际宽度，ymax - ymin 的值是图像的实际高度，单位是“像素”，大多数情况下，xmin 和 ymin 的值都是 0。hres 和 vres 是图像的水平和垂直分辨率。palette 是长度为 48 字节的彩色调色板，它只用于 16 色以下图像使用。byte_per_line 代表图像解码后每一行图像数据所需的字节数，图像解码软件可直接只用这个值分配解码缓冲区，省去重新计算的麻烦。

PCX 文件的第二部分是采用 PCX_RLE 算法压缩的图像数据，这也是 RLE 算法的一个变种，下一节会介绍这种算法的原理。PCX 文件的最后一部分是 256 色图像才有的 256 色调色板数据，由 1 字节的颜色数和 768 字节的调色板数据组成。之所以要附加这一部分数据，是因为在定义 PCX 文件头时，没有考虑到以后的图像会有这么多颜色，因此只预留了 48 字节的调色板数据区，这显然无法存放超过 16 种颜色的调色板数据，因此只好在 PCX 文件的尾部附加了这样一块数据区。并不是所有 PCX 文件都有 256 色调色板，对于 256 色调色板数据，要结合文件头部的 version 属性和 bits_per_pixel 属性使用。

10.2.2 PCX_RLE 算法

10.1 节介绍过，对 RLE 算法的优化主要集中在对连续非重复数据的处理方式上，各种优化方法的差别则体现在如何区分连续重复数据和连续非重复数据上。PCX_RLE 算法根据长度属性字节的最高位和次高位来标识这个标志，对于连续非重复数据则不使用任何标识字节。也就是说，如果一个数据的最高两个比特都是 1，则说明这是一个长度属性字节，其低 6 位表示重复数据的长度，如果最高位两位不是 1，则表示这个字节本身就是非重复数据。这样一来就引入了一个问题，如果非重复数据本身刚好最高两位是 1，就会与连续重复数据的标志产生冲突。PCX_RLE 算法解决这个问题的方法是将这样的数据当作长度为 1 的连续重复数据处理，也就是在数据前插入一个 0xC1 标识。

```
int PcxRle_Encode(unsigned char *inbuf, int inSize, unsigned char *outbuf, int onuBufSize)
{
    unsigned char *src = inbuf;
    int i;
    int encSize = 0;

    while(src < (inbuf + inSize))
    {
        unsigned char value = *src++;
        i = 1;
```

```
            while((*src == value) && (i < 63))
            {
                src++;
                i++;
            }

            if(i > 1)
            {
                outbuf[encSize++] = i | 0xC0;
                outbuf[encSize++] = value;
            }
            else
            {
                /*如果非重复数据最高两位是，插入标识字节*/
                if((value & 0xC0) == 0xC0)
                {
                    outbuf[encSize++] = 0xC1;
                }
                outbuf[encSize++] = value;
            }
        }

        return encSize;
    }
```

PcxRle_Encode()函数是 PCX_RLE 算法的编码函数，对于颜色比较简单的绘图文件，颜色数一般不多，加上对调色板的优化，可以使大部分数据避开最高两个比特是 1 的情况，应用这个算法的 PCX 文件还是具有很好的压缩效果的。PCX_RLE 算法的解码函数非常简单，在下一节处理 PCX 文件数据的代码中，可以看到解码算法的体现。

10.2.3 256 色 PCX 文件的解码和显示

256 色 PCX 文件的图像数据并不是对应点的颜色，而是对应点的颜色在调色板中的索引，需要根据这个索引从调色板中查到真正的颜色才能显示图像。由于不需要处理与显示设备相关的颜色位平面问题，256 色 PCX 文件的解码和显示算法非常简单。首先从文件开始位置读取 128 字节的文件头结构，判断是否是 256 色 PCX 文件，然后从文件尾部向前偏移 769 字节，读取颜色数和调色板数据，最后定位到图像数据部分，逐行解码并显示图像数据。假设已经有 bool GetPcxfileHeader(FILE *fp, PCX_HEAD *header) 函数负责读取并检查 PCX 文件头，void DrawPixel(int x, int y, int colorIdx)函数负责根据颜色索引和调色板数据在显示设备的[*x*,*y*]位置画一个点，则处理显示 PCX 文件的算法框架如下：

```
unsigned char *bitsLine = new unsigned char[header.bytes_per_line];
int height = header.ymax - header.ymin;
int width = header.xmax - header.xmin;
int srcIdx = 0;
for(int y = 0; y < height; y++)
{
    srcIdx += DecodePcxLine(&header, imgData + srcIdx, bitsLine);
```

```
        for(int x = 0; x < width; x++)
        {
            DrawPixel(x, y, bitsLine[x]);
        }
    }
```

DecodePcxLine()函数就是 PCX_RLE 算法的核心，imgData+srcIdx 是压缩数据，解压缩后的数据存放在 bitsLine 指向的缓冲区中。由 header 参数中的 bytes_per_line 属性控制，每次解压缩一行图像数据。

```
int DecodePcxLine(PCX_HEAD *header, unsigned char *imgData, unsigned char *lineBuf)
{
    int i = 0;
    int offset = 0;
    while(i < header->bytes_per_line)
    {
        unsigned char value = imgData[offset++];
        if((value & 0xC0) == 0xC0) /*判断标志*/
        {
            value = value & 0x3F; /*count*/
            for(int repeat = 0; repeat < value; repeat++)
            {
                lineBuf[i++] = imgData[offset];
            }
            offset++;
        }
        else
        {
            lineBuf[i++] = value;
        }
    }

    return offset;
}
```

看到了吧，PCX 文件的解码显示就是这么简单，DOS 时代的很多软件都使用 PCX 文件作为 UI 界面的点缀。我的第一个 DOS 图形界面软件用一张荷花的照片作为 Splash 窗口，用的就是 PCX 文件，没有别的原因，就是简单。

10.3 总结

本章介绍了一种简单的数据压缩算法，看似简陋的算法，在很多情况下却有非常好的效果。曾经流行一时的 PCX 文件采用 RLE 算法作为数据压缩算法，说明 RLE 算法在颜色相对单调的图像数据处理中具有非常好的适用性。BMP 是另一种在 Windows 平台上非常流行的图像文件格式，BMP 文件也支持两种形式的压缩格式，分别称为 BI_RLE8 和 BI_RLE4，这也是两种 RLE 算法的变种算法，在处理单色位图和背景比较单调的图像时采用这两种压缩方式，也可以取得很不错的效果。

10.4 参考资料

[1] 维基百科“PCX 文件”：http://zh.wikipedia.org/wiki/PCX
[2] 维基百科“游程编码”：http://zh.wikipedia.org/wiki/RLE

第 11 章 算法与历法

日历在我们的生活中扮演着十分重要的角色，上班、上学、约会都离不开日历。每当新的一年开始时，人们都要更换新的日历。你想知道未来一年的这么多天是怎么确定下来的吗？为什么 2014 年的国庆节是星期三而 2015 年的国庆节是星期四？为什么每年的农历春节都相差那么多天？闰五月真的能过两次端午节吗？那就来研究一下日历算法吧。本章就来介绍日历的编排规则与算法。这里面既有简单的算法，比如确定某日是星期几的计算方法，以及打印公历年历的算法，也有复杂的算法，比如利用天文算法精确计算二十四节气和日月合朔时间的算法，这些是推算中国农历的基础算法。最后我们实现一个可以显示公历和农历双历的日历控件，通过这个日历控件的演示程序介绍将公历和农历合成双历的对照算法。

11.1 格里历（公历）生成算法

从上小学开始，我就喜欢数学课胜过喜欢语文课，我到现在还记得最有意思的一节数学课是我们的数学老师带着我们做日历。在知道了新的一年第一天是星期几之后，我们就开始利用简单的历法规则推算之后的每一天是星期几，每个月有几天，并最终画一张新年年历，很多同学甚至根据闰年规律推算出了今后几十年的日历。这是一个非常有意思的问题，其实就是关于日历的生成算法。

要研究日历算法，首先要知道日历的编排规则，也就是历法。所谓历法，就是推算年、月、日的时间长度和它们之间的关系，指定时间序列的法则。我国的官方历法是目前全球各国通用的公历，也就是公元纪年。公历实际上是从 1582 年 10 月 15 日开始实行的格里历（Gregorian Calendar）。这是一个比较简单的历法，有人称之为“规范历”。所谓的“规范”，言外之意其实就是简单。生成格里历的日历算法也相对简单，需要特殊处理的仅仅是星期的问题。

11.1.1 格里历的历法规则

格里历的历法规则非常简单，它首先将年份分成平常年（或平年，Common Year）和闰年（Leap

Year）两种。格里历的一年分为十二个月，其中一月、三月、五月、七月、八月、十月和十二月是大月，大月的一个月有 31 天。四月、六月、九月和十一月是小月，小月的一个月有 30 天。二月天数要根据是否是闰年来定，如果是闰年，二月是 29 天，如果是平常年，二月是 28 天。平常年一年是 365 天，闰年一年是 366 天，判定一年是平常年还是闰年的规则如下：

(1) 如果年份是 4 的倍数，且不是 100 的倍数，则是闰年；

(2) 如果年份是 400 的倍数，则是闰年；

(3) 不满足(1)、(2)条件的就是平常年。

格里历的置闰规则简单总结就是一句话：四年一闰，百年不闰，四百年再闰。为什么格里历会有这么奇怪的规则？看了 11.4.1 节的介绍你就明白了，这里你只需要记住这句话就行了。判断给定的年份是否是闰年的算法也是一个很经典的算法，结合了与、非等逻辑判断和组合，是面试常见的问题之一。

```
bool IsLeapYear(int year)
{
    return ((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0);
}
```

这就是格里历的历法规则，如果不考虑星期的问题，这个历法真是一点乐趣都没有。决定今天是星期几虽然不是格里历历法的内容，但是是人们生活必需的内容，下面我们就来讨论一下今天到底是星期几的问题。

11.1.2 今天星期几

除了年、月、日，人们日常生活中还对日期定义了另一个属性，就是星期几。星期并不是格里历范畴内的东西，但是人们已经习惯用星期来管理和规划时间，比如一个星期工作五天，休息两天，等等。星期的规则彻底改变了人们的生活习惯，因此星期已经成为历法的一部分了。星期的命名最早起源于古巴比伦。公元前 7～6 世纪，巴比伦人就使用了星期制，一个星期中的每一天都有一个天神掌管。这一制度后来传到古罗马，并逐渐演变成现在的星期制度。

如何知道某一天到底是星期几？除了查日历之外，是否有办法推算出来某一天是星期几呢？答案是肯定的，星期不像年和月那样有固定的历法规则，但是星期的计算也有自己的规律。星期是固定的 7 天周期，其排列顺序固定，不受闰年、平常年以及大小月的天数变化影响。因此，只要确切地知道某一天是星期几，就可以推算出其他日期是星期几。推算的方法很简单，就是计算两个日期之间相差多少天，用相差的天数对 7 取余数，这个余数就是两个日期的星期数的差值。举个例子，假设已经知道 1977 年 3 月 27 日是星期日，如何得知 1978 年 3 月 27 日是星期几？按照前面的方法，计算出 1977 年 3 月 27 日到 1978 年 3 月 27 日之间相差 365 天，365 除以 7 余数是 1，所以 1978 年 3 月 27 日就是星期一。

上述方法计算星期几的关键是求出两个日期之间相隔的天数。有两种常用的方法计算两个日期之间相隔的天数，一种是利用公历的月和年的规则直接计算，另一种是利用儒略日计算。除此

之外，还可以利用蔡勒（Zeller）公式计算某一天是星期几，下面就分别介绍一下这三种常用的方法。

1. 直接根据日期的差值

利用公历规则直接计算两个日期之间相差的天数，最简单的方法就是将两个日期之间相隔的天数分成三个部分：前一个日期所在年份还剩下的天数、两个日期之间相隔的整数年所包含的天数以及后一个日期所在的年过去的天数。分别计算这三个部分的天数然后求和得到两个日期最终相差的天数。如果两个日期是相邻两个年份的日期，则第二部分整年的天数就是 0。以 1977 年 3 月 27 日到 2005 年 5 月 31 日为例，1977 年还剩下的天数是 279 天，中间整数年是从 1978 年到 2005 年（不包括 2005 年），共 27 年，包括 7 个闰年和 20 个平常年，总计 9862 天，最后是 2005 年从 1 月 1 日到 5 月 31 日经过的天数 151 天。三者总和是 10292 天。10292 除以 7 的余数是 2，已知 1977 年 3 月 27 日是星期日，因此 2005 年 5 月 31 日是星期二。这个计算的算法实现并不难，只要细心处理好闰年以及大小月的关系，一般不会出错。

2. 利用儒略日计算日期的差值

另一种计算两个日期相差天数的方法是利用儒略日（Julian Day，JD）进行计算。首先介绍一下儒略日，儒略日是一种不记年、不记月、只记日的历法，是由法国学者 Joseph Justus Scaliger（1540—1609）在 1583 年提出来的一种以天数为计量单位的流水日历。儒略日和儒略历（Julian Calendar）没有任何关系，命名为儒略日仅仅是因为他本人为了纪念他的父亲——意大利学者 Julius Caesar Scaliger（1484—1558）。简单来讲，儒略日就是指从公元前 4713 年 1 月 1 日 UTC 12:00 开始所经过的天数，JD0 就被指定为公元前 4713 年 1 月 1 日 12:00 到公元前 4713 年 1 月 2 日 12:00 之间的 24 小时，以此顺推，每一天都被赋予一个唯一的数字。例如从 1996 年 1 月 1 日 12:00 开始的一天就是儒略日 JD2450084。使用儒略日可以把不同历法的年表统一起来，很方便地在各种历法中追溯日期。需要注意的是，儒略日并不是只关注天数，它是一个浮点数，可以精确到秒，一秒钟对应的儒略日差值是 0.0000115740 个儒略日。另一个需要注意的是儒略日并不是从 0:00 开始的，它是从中午 12:00 开始的 24 个小时，因此在计算日期时如果需要考虑 0:00 开始的关系，需要增加或减少 0.5 个儒略日进行修正。

如果计算两个日期之间的天数，利用儒略日计算也很方便，先计算出两个日期的儒略日数，然后直接相减就可以得到两个日期相隔的天数。由格里历的日期计算出儒略日数是一个很简单的事情，有多个公式可以计算儒略日，本书选择如下公式计算儒略日：

$$JD = \left\lfloor \frac{153m+2}{5} \right\rfloor + 365y + \left\lfloor \frac{y}{4} \right\rfloor - \left\lfloor \frac{y}{100} \right\rfloor + \left\lfloor \frac{y}{400} \right\rfloor + day - 32045 \tag{11-1}$$

$$JD = \left\lfloor \frac{153m+2}{5} \right\rfloor + 365y + \left\lfloor \frac{y}{4} \right\rfloor + day - 32083 \tag{11-2}$$

式(11-1)适用于格里历，式(11-2)适用于正式启用格里历之前所使用的儒略历。关于儒略历和

格里历的历史，请参见 11.1.4 节的介绍。上式中的 y 和 m 可用以下公式计算：

$$a = \left\lfloor \frac{14 - month}{12} \right\rfloor$$

$$y = year + 4800 - a$$

$$m = month + 12a - 3$$

以上各式中，*year*、*month* 和 *day* 分别是对应日历日期中的年份、月份和日期。需要注意的是，这个公式求出的结果是某日正午 12:00（标准时间）对应的儒略日，如果只求解整数精度的儒略日，直接使用这个公式即可。如果要求解精确到时分秒的儒略日，需要对其做 – 0.5 个儒略日的修正，具体实现请看以下计算儒略日的算法实现。

```
double CalculateJulianDay(int year, int month, int day, int hour, int minute, double second)
{
    int a = (14 - month) / 12;
    int y = year + 4800 - a;
    int m = month + 12 * a - 3;

    double jdn = day + (153 * m + 2) / 5 + 365 * y + y / 4;
    if(IsGregorianDays(year, month, day))
    {
        jdn = jdn - y / 100 + y / 400 - 32045.5;
    }
    else
    {
        jdn -= 32083.5;
    }

    return jdn + hour / 24.0 + minute / 1440.0 + second / 86400.0;
}
```

1977 年 3 月 27 日 12:00 的儒略日是 JD2443230.0，2005 年 5 月 31 日 12:00 的儒略日是 JD2453522.0，差值是 10292，与直接利用公历规则计算的差值一致。实际上，既然儒略日是相对于公元前 4713 年 1 月 1 日开始的流水日历，而公元前 4713 年 1 月 1 日又被指定为星期一，所以直接对儒略日整数部分取余，也可以推算出这一天是星期几。比如 2005 年 5 月 31 日 12:00 的儒略日是 JD2453522.0，对 7 取余得到的结果是 1，也就是说 2005 年 5 月 31 日与公元前 4713 年 1 月 1 日星期数差 1，那 2005 年 5 月 31 日自然就是星期二了。

3. 利用蔡勒公式计算星期数

上述计算星期的方法虽然步骤简单，但是每次都要计算两个日期的时间差，不是非常方便。如果能够有一个公式可以直接根据日期计算出对应的星期岂不是更好？幸运的是，这样的公式是存在的。此类公式的推导原理仍然是通过两个日期的时间差来计算星期，只是通过选择一个特殊的日期来简化公式的推导。这个特殊日期指的是某一年的 12 月 31 日这天刚好是星期日这种情况。选择这样的日子有两个好处，一是计算上可以省去计算标准日期这一年的剩余天数，二是计算出

来的日期差余数是几就是星期几，不需要再计算星期的差值。现在我们就来推导一个这样的求星期几的公式。

我们知道公元元年的 1 月 1 日是星期一，那么公元前 1 年的 12 月 31 日就是星期日，用这一天作为标准日期，就可以只计算整数年的时间和日期所在的年积累的天数。这个星期公式如下：

$$w = (L \times 366 + N \times 365 + D)\%7 \tag{11-3}$$

公式中的 L 是从公元元年到 $year$ 年 $month$ 月 day 日所在的年之间发生闰年的次数，N 是平常年的次数，D 是 $year$ 年内的积累天数。将整年数 $year - 1 = L + N$ 带入上式，可得：

$$w = ((year - 1) \times 365 + L + D)\%7 \tag{11-4}$$

根据闰年规律，从公元元年到 y 年之间的闰年次数是可以计算出来的，即：

$$L = \left\lfloor \frac{year - 1}{4} \right\rfloor - \left\lfloor \frac{year - 1}{100} \right\rfloor + \left\lfloor \frac{year - 1}{400} \right\rfloor \tag{11-5}$$

将式(11-5)代入到式(11-4)，得到最终的计算公式：

$$w = ((year - 1) \times 365 + \left\lfloor \frac{year - 1}{4} \right\rfloor - \left\lfloor \frac{year - 1}{100} \right\rfloor + \left\lfloor \frac{year - 1}{400} \right\rfloor + D)\%7 \tag{11-6}$$

仍然以 2005 年 5 月 31 日为例，利用公式(11-6)计算 w 的值为：

$$w = ((2005 - 1)*365 + \lfloor (2005-1)/4 \rfloor - \lfloor (2005-1)/100 \rfloor + \lfloor (2005-1)/400 \rfloor + 151)\%7$$
$$= (731460 + 501 - 20 + 5 + 151)\ \%\ 7 = 732097\ \%\ 7 = 2$$

得到 2005 年 5 月 31 日是星期二，和前面的计算方法得到的结果一致。公式(11-6)的问题在于计算量大，不利于口算星期结果。于是人们就在式(11-6)的基础上继续推导更简单的公式。德国数学家克里斯蒂安·蔡勒（Christian Zeller, 1822—1899）在 1886 年推导出了著名的为蔡勒公式：

$$w = (y + \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{c}{4} \right\rfloor - 2c + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + d - 1)\%7 \tag{11-7}$$

最后计算出的余数 w 是几，结果就是星期几，如果余数是 0，则为星期日。蔡勒公式中各符号的含义如下。

c：世纪数–1 的值，如 21 世纪，则 $c = 20$。

m：月数，m 的取值是大于等于 3，小于等于 14。在蔡勒公式中，某年的 1 月和 2 月看作上一年的 13 月和 14 月，比如 2001 年 2 月 1 日要当成 2000 年的 14 月 1 日计算。

y：年份，取公元纪念的后两位，如 1998 年，$y = 98$，2001 年，$y = 1$。

d：某月内的日数

为了方便口算，人们通常将公式中的$\lfloor 13(m+1)/5 \rfloor$一项改成$\lfloor 26(m+1)/10 \rfloor$。目前人们普遍认为蔡勒公式是计算某一天是星期几的最好的公式。但是蔡勒公式有时候计算出的结果可能是负数，需要对结果+7 进行修正。比如 2006 年 7 月 1 日，用蔡勒公式计算出的结果是 –1，实际上这天是星期六。

记得有一次看到电视上介绍一个牛人，号称记忆力惊人，可以记得任何一天是星期几。只要记住蔡勒公式，心算快一点，你就也是牛人了。ZellerWeek()函数就是蔡勒公式的算法实现，注意对月份的修正以及最后结果为负数的修正，其他的内容就是将蔡勒公式翻译成代码。

```
int ZellerWeek(int year, int month, int day)
{
    int m = month;
    int d = day;

    if(month <= 2) /*对小于的月份进行修正*/
    {
        year--;
        m = month + 12;
    }

    int y = year % 100;
    int c = year / 100;

    int w = (y + y / 4 + c / 4 - 2 * c + (13 * (m + 1) / 5) + d - 1) % 7;
    if(w < 0) /*修正计算结果是负数的情况*/
        w += 7;

    return w;
}
```

蔡勒公式和前面提到的式(11-6)都只适用于格里历法。罗马教皇在 1582 年修改历法，将 10 月 5 日指定为 10 月 15 日，从而正式废止儒略历法，开始启用格里历法。因此，上述求星期几的公式只适用于 1582 年 10 月 15 日之后的日期，对于 1582 年 10 月 4 日之前的日期，蔡勒也推导出了适用于儒略历法的星期计算公式，如式(11-8)所示，有兴趣的读者可自行完成算法实现。

$$w = (5 - c + y + \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + d - 1)\%7 \tag{11-8}$$

式(11-8)适用于对 1582 年 10 月 4 日之前的日期计算星期，1582 年 10 月 5 日与 1582 年 10 月 15 日之间的日期是不存在的，因为它们都是同一天。

11.1.3 生成日历的算法

日历一般以月为单位组织，生成日历需要知道每个月有多少天。格里历历法简单，除二月外每月天数固定，二月则根据是否是闰年确定是 28 天还是 29 天，比较适合使用查表法实现。首先定义一个 daysOfMonth 表，再次利用数组下标的技巧，直接用其表示月份：

```
int daysOfMonth[MONTHES_YEAR] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

然后轻松给出算法实现：

```
int GetDaysOfMonth(int year, int month)
{
    if((month < 1) || (month > MONTHES_YEAR))
        return 0;

    int days = daysOfMonth[month - 1];
    if((month == 2) && IsLeapYear(year))
    {
        days++;
    }

    return days;
}
```

确定每个月的天数之后，就可以依次排列输出这个月的所有日期。星期的位置是固定的，每个月的第一天是星期几就从星期几对应的位置开始排列数字，遇到星期六（星期的位置是从星期日到星期六排列）就折返下一行继续输出。每个月第一天的星期数可以用蔡勒公式计算，之后的每一天不必重复使用蔡勒公式，用 `week = (week + 1) % 7` 直接推算就可以了。

```
void PrintMonthCalendar(int year, int month)
{
    int days = GetDaysOfMonth(year, month); /*确定这个月的天数*/
    int firstDayWeek = ZellerWeek(year, month, 1);
    InsertRowSpace(firstDayWeek);
    int week = firstDayWeek;
    int i = 1;
    while(i <= days)
    {
        printf("%-10d", i);
        if(week == 6) /*到一周结束，切换到下一行输出*/
        {
            printf("\n");
        }
        i++;
        week = (week + 1) % 7;
    }
}
```

`PrintMonthCalendar()`函数打印指定年和月的月历，如上所述，都是非常简单的算法。`InsertRowSpace()`函数负责根据每个月第一天的星期数插入合适的空格位置，使每个月的第一天与实际的星期位置能对上。对每个月依次调用 `PrintMonthCalendar()`函数即可打印出一年的日历。想想那一节数学课做的事情，当初要是有这个算法就省事多了。

11.1.4 日历变更那点事儿

中国古代历朝历代都要修订历法，历法那就是王法，谁当王谁说了算，直到汉唐以后，才形

成了比较稳定的农历历法。西方人也经常干这种事情，最近的一次变更就是在 1582 年启用格里历。格里历是罗马教皇格里十三世颁布实施的，但是整个欧洲教会也不是铁板一块，各国皇室都不甩教皇那一套。德国和荷兰直到 1698 年才使用格里历，距离格里历的颁布已经过去一百多年了；而英国则在 1752 年才由议会批准使用格里历；至于沙皇俄国，直到 1918 年革命才开始使用格里历，比中国还晚。研究这个时期欧洲各国的历史是一件十分头疼的事情，同一个事件在不同国家的文献记载中发生的时间也不相同，研究者必需时时留意各国历法变更的情况。

本节我们讨论一下从儒略历到格里历变更的事情，同时解释一下格里历为什么会有这么奇怪的置闰规则。

1. 儒略历和格里历

在公元 1582 年 10 月 15 日之前，人们使用的历法是源自古罗马的儒略历。儒略历的置闰规则非常简单，就是四年一闰。这种置闰规则使得历法时间每年比天文时间多出来 0.0078 天，这样从公元前 46 年到公元 1582 年一共累计多出了 10 天。再这样下去历法和天气时节就要脱节了，为此，当时的教皇格里十三世将 1582 年 10 月 5 日人为指定为 10 月 15 日，并开始启用新的置闰规则，这就是后来沿用至今的格里历。

2. 1752 年 9 月到底是怎么回事儿

如果你用的操作系统是 Unix 或 Linux，在控制台输入以下命令：

```
#cal 9 1752
```

你会看到这样一个奇怪的月历输出：

```
September 1752
Su Mo Tu We Th Fr Sa
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

1752 年的 9 月缺了 11 天，到底怎么回事儿？这其实还是因为从儒略历到格里历的转换造成的。1582 年 10 月 5 日，罗马教皇格里十三世宣布启用更为精确的格里历，但是整个欧洲大陆并不是所有国家都立即采用格里历，比如英国就是直到 1752 年 9 月议会才批准采用格里历，所以英国及其所有殖民地的历法一直到 1752 年 9 月才发生跳变，“跟上”了格里历。Linux 的 `cal` 指令起源于最初 AT&T 的 Unix，当然采用的是美国历法，但是美国历史太短，再往前就只能采用英国历法，所以 `cal` 指令的结果就成了这样。对于采用格里历的国家来说，只要知道 1582 年 10 月发生了日期跳变就行了，可以不用关心 1752 年 9 月到底是怎么回事儿。但是对于研究历史和考古的人来说，就必需要了解这段历史，搞清楚每个欧洲国家改用格里历的年份，否则就可能在一些问题上出错。在欧洲研究历史，你会发现很多事件都是有多个时间版本的，比如大科学家牛顿的生日就有两个时间版本，一个是按照儒略历历法的 1642 年 12 月 25 日，另一个是格里历历法的 1643 年 1 月 4 日。对于英国人来说，1752 年之前都是按照儒略历计算的，所以英国的史书可能会记载牛顿出生在圣诞节，这也没什么可奇怪的。

3. 公历的闰年

格里历的置闰规则是“四年一闰、百年不闰、四百年再闰”，为什么会有这么奇怪的置闰规则呢？这实际上与天体运行周期与人类定义的历法周期之间的误差有关。地球绕太阳运转的周期是 365.2422 天，即一个回归年（tropical year），而公历的一年是 365 天，这样一年就比回归年少了 0.2422 日，四年积累下来就少了 0.9688 天（约 1 天），于是设置一个闰年，让这一年多一天。这样一来，四个公历年又比四个回归年多了 0.0312 天，这样经过四百年就会多出 3.12 天，也就是说每四百年要减少 3 个闰年才行，于是就设置了“百年不闰、四百年再闰”的置闰规则。

实际上公历的置闰还有一条规则，就是对于数值很大的年份，如果能整除 3200，还必须同时能整除 86400 才是闰年。这是因为前面即使四百年一闰，仍然多了 0.12 天，平均就是每年多 0.0003 天，于是每 3200 年就又多出 0.96 天，于是能被 3200 整除的年就不是闰年了。然而误差并没有终结，每 3200 年减少一个闰年（减少一天）实际上多减了 0.04 天，这个误差还要继续累计计算，只要凑 24 个 3200 年周期，发现又凑出了 0.96 天，于是可以设置闰年了，这就是每 3200 年不闰，86400 年再闰的原因。但是你注意没有，这个误差还是存在，还需要继续凑。我已经崩溃了，读者有兴趣自己算吧。最后的置闰规则是这样的：能被 4 整除且不能被 100 整除；或者能被 400 整除且不能被 3200 整除；或者能被 86400 整除的年份是闰年。

是谁说公历是精确的历法？看到这个置闰规则后就不会有人再这么说了吧？只要人们采用“天”作为历法单位，就不会有精确的历法。天体的运行规律怎么可能刚好合乎人类要求？假如有一天，人类给地球加装一个推进装置，能精确控制地球公转的周期刚好是 365 天，这样人类就有精确的历法了。要不，不要用天计时了，全都用秒，比如“五千亿四千八百万三千三百二十一宇宙秒时，我在电影院门口等你，不见不散”。你想过这样的生活吗？还是老老实实用格里历吧。

11.2 二十四节气的天文学计算

中国古代历法都是以月亮运行规律为主，严格按照朔望月长度定义月，但是由于朔望月长度和地球回归年长度无法协调，会导致农历季节和天气的实际冷暖无法对应，因此聪明的古人将月亮运行规律和太阳运行规律相结合制定了中国农历的历法规则。在这种特殊的阴阳结合的历法规则中，二十四节气就扮演着非常重要的作用，它是联系月亮运行规律和太阳运行规律的纽带。正是由于二十四节气结合置闰规则，使得农历的春夏秋冬四季和地球绕太阳运动引起的天气冷暖变化相一致，成为中国几千年来生产和生活的依据。

二十四节气在中国古代历法中扮演着非常重要的角色，本节将介绍二十四节气的基本知识，以及如何使用 VSOP82/87 行星运行理论计算二十四节气发生的准确时间。

11.2.1 二十四节气的起源

二十四节气起源于中国黄河流域。远在春秋时代，古人就开始使用仲春、仲夏、仲秋和仲冬四个节气指导农耕种植。后来经过不断地改进与完善，到秦汉年间，二十四节气已经基本确立。

公元前 104 年（汉武帝太初元年），汉武帝颁布由邓平等人制定的《太初历》，正式把二十四节气订于历法，明确了二十四节气的天文位置。二十四节气天文位置的定义，就是从太阳黄经零度开始，沿黄经每运行 15 度所经历的时日称为一个节气。太阳一个回归年运行 360 度，共经历二十四个节气，每个公历月对应两个节气。其中，每月第一个节气为“节令”，即立春、惊蛰、清明、立夏、芒种、小暑、立秋、白露、寒露、立冬、大雪和小寒十二个节令；每月的第二个节气为“中气”，即雨水、春分、谷雨、小满、夏至、大暑、处暑、秋分、霜降、小雪、冬至和大寒十二个中气。“节令” 和“中气”交替出现，各历时 15 天，人们习惯上把“节令”和“中气”统称为“节气”。

11.2.2　二十四节气的天文学定义

为了更好地理解二十四节气的天文位置，首先要解释几个天文学概念。“天球”是人们为了研究天体的位置和运动规律而引入的一个假象的球体，根据观察点（也就是球心）的位置不同，可分为“日心天球”、“地心天球”等。图 11-1 就是天球概念的一个简单示意图。

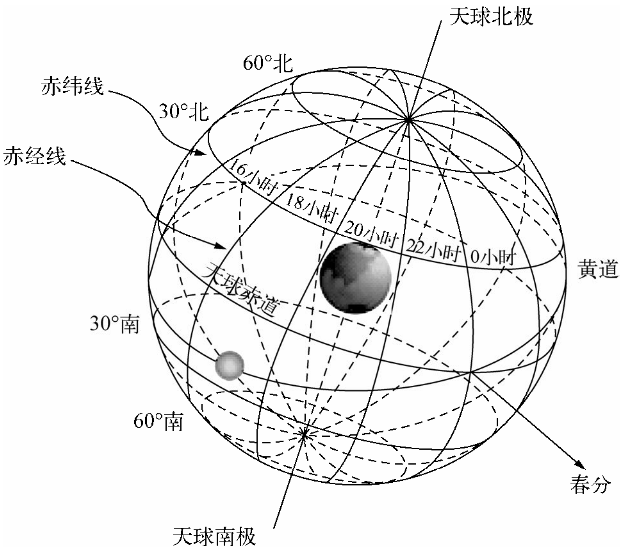


图 11-1　天球概念示意图

天文学中常用的一个坐标体系就是“地心天球”，它与地球同心且有相同的自转轴，理论上具有无限大的半径。地球的赤道和南北极点延伸到天球上，对应着天赤道和南北天极点。和地球上用经纬度定位位置一样，天球也划分了经纬度，分别命名为“赤经”和“赤纬”，地球上的经度以度（分秒）为单位，赤经以时（分秒）为单位。天空中的所有天体都可以投射到天球上，用赤经和赤纬定位天体在天球上的位置。地球沿着一个近似椭圆的轨道绕太阳公转，这个公转轨道所在的平面就是“黄道面”。“黄道”（ecliptic）是地球绕太阳公转轨道所在的“黄道面”向外延

伸与天球（地心天球）相交的大圆，由于地球公转受月球和其他行星的摄动，地球的公转轨道并不是严格的平面，因此黄道的严格定义是：地月系质心绕太阳公转的瞬时平均轨道平面与天球相交的大圆。黄道和天赤道所在的两个平面并不是重叠的，它们之间存在一个 23 度 26 分的交角，称为“黄赤交角”。由于黄赤交角的存在，黄道和天赤道就在天球上有两个交点，这两个交点就是春分点和秋分点。在天球上以黄道为基圈可以形成黄道坐标系，在黄道坐标系中，也使用了经纬度的概念，分别称为“黄经”和“黄纬”。天体的黄经从春分点起沿黄道向东计量，春分点是黄经 0 度，沿黄道一周是 360 度，使用的单位是度、分和秒。黄纬以黄道测量平面为准，向北记为 0 度到 90 度，向南记为 0 度到–90 度。

黄道平面可以近似地理解为地球绕太阳公转的平面，以黄道为基圈的黄道坐标系根据观测中心是太阳还是地球还可以区分为日心坐标系和地心坐标系，对应天体的黄道坐标分别被称为“日心黄经、日心黄纬”和“地心黄经、地心黄纬”。日心黄经和日心黄纬比较容易理解，因为太阳系的行星都是绕太阳公转的，以太阳为中心将这些行星向天球上投影是最简单的确定行星位置关系的做法。但是人类自古观察太阳的周年运动，都是以地球为参照，以太阳的周年视运动位置来计算太阳的运行轨迹，使用的其实都是地心黄经和地心黄纬。要了解古代历法，理解这一点非常重要。图 11-2 就解释了造成这种视觉错觉的原因。古人由于观测条件限制，只能根据视觉感觉认为是太阳沿着黄道绕地球运转，因此设定太阳从黄经（黄道经度）零度起（以春分点为起点自西向东度量），将太阳沿黄经每运行 15 度所经历的时日称为一个节气。太阳每年运行 360 度，共经历二十四个节气，春季的节气有立春（315 度）、雨水（330 度）、惊蛰（345 度）、春分（0 度、360 度）、清明（15 度）和谷雨（30 度），夏季的节气有立夏（45 度）、小满（60 度）、芒种（75 度）、夏至（90 度）、小暑（105 度）和大暑（120 度），秋季的节气有立秋（135 度）、处暑（150 度）、白露（165 度）、秋分（180 度）、寒露（195 度）和霜降（210 度）。冬季的节气有立冬（225 度）、小雪（240 度）、大雪（255 度）、冬至（270 度）、小寒（285 度）和大寒（300 度）。二十四个节气平分在公历的 12 个月中，每月一节气一中气。二十四节气反映了太阳的周年运动（以地球为参照物的视运动），所以节气在现行的公历中日期基本固定，上半年在 6 日、21 日，下半年在 8 日、23 日，前后不差 1 ~ 2 天。中国民间流传的《二十四节气歌》就是为了方便记忆这些节气。

春雨惊春清谷天，
夏满芒夏暑相连，
秋处露秋寒霜降，
冬雪雪冬小大寒，
每月两节不变更，
最多相差一两天。

古人定义二十四节气的位置，是太阳沿着黄道运行时的视觉位置，每个节气对应的黄道经度其实是地心黄经。从图 11-2 可以看出，日心黄经和地心黄经存在 180 度的转换关系，同样可以理解，日心黄纬和地心黄纬在方向上是相反的，因此可以很方便地将两类坐标相互转换，转换公

式是：

$$太阳地心黄经 = 地球日心黄经 + 180° \quad (11\text{-}9)$$

$$太阳地心黄纬 = -地球日心黄纬 \quad (11\text{-}10)$$

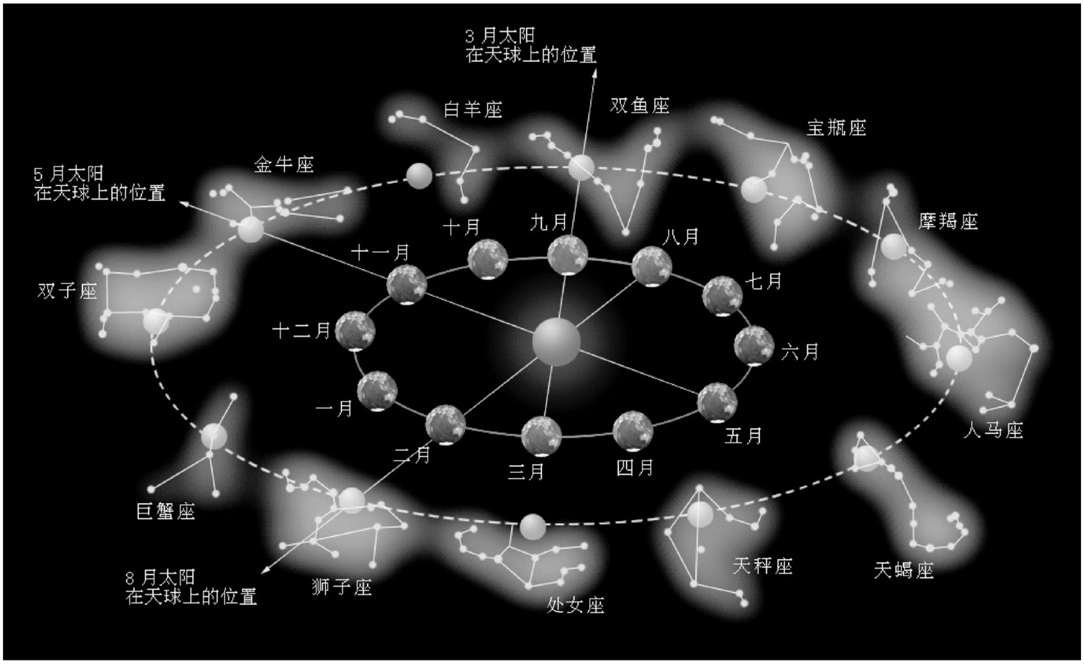


图 11-2　太阳黄道视觉位置原理图（图片来自百度百科）

了解了以上天文学基础之后，就可以着手对二十四节气的发生时间进行计算了。我们常说的节气发生时间，其实就是在太阳沿着黄道做视觉运动的过程中，当太阳地心黄经等于某个节气黄经度数时的那个瞬间的时间。所谓的用天文算法计算二十四节气时间，就是根据牛顿力学原理或开普勒三大行星定律，计算出与历法密切相关的地球、太阳和月亮三个天体的运行轨道和时间参数，以此得出当这些天体位于某个位置时的时间。这样的天文计算需要计算者有扎实的微积分学、几何学和球面三角学知识，令广大天文爱好者望而却步。但是随着 VSOP-82/87 行星理论以及 ELP-2000/82 月球理论的出现，天文计算变得简单易行。

11.2.3　VSOP-82/87 行星理论

古代天文学家在对包括地球和月亮在内的行星运行轨道精确计算后发现，天体的运行因为受相近天体的影响，并不严格遵循理论方法计算出来的轨道，而是在理论轨道附近波动。这种影响在天文学上称为摄动，摄动很难精确计算，只能根据经验估算。但是经过长期的观测和计算，天文学家发现行星轨道因为摄动影响而产生的波动其实也是有规律的，即在相当长的时间内呈现出

周期变化的趋势。于是天文学家开始研究这种周期变化，希望通过一种类似曲线拟合的方法，对一些周期计算项按照某种计算式迭代求和计算代替积分计算来模拟行星运行轨迹。这种计算式可以描述为：$a + bt + ct2 + \cdots x\cos(p + qt + rt2 + \cdots)$，其中 t 是时间参数，这样的理论通常称为半解析（semi-analytic）理论。其实早在 18 世纪，欧洲学者 Joseph Louis Lagrange 就开始尝试用这种周期项计算的方法修正行星轨道，但是他采用的周期项计算式是线性方程，精度不高。

1982 年，P. Bretagnon 公开发表了 VSOP 行星理论①，该理论是一个描述太阳系行星轨道在相当长时间范围内周期变化的半解析理论。VSOP82 理论是 VSOP 理论的第一个版本，提供了对太阳系几大行星位置计算的周期序列，通过对周期序列进行正弦或余弦项累加求和，就可以得到这个行星在给定时间的轨道参数。不过 VSOP82 由于每次都会计算出全部超高精度的轨道参数，这些轨道参数对于历法计算这样的民用场合很不适用。1987 年，Bretagnon 和 Francou 创建了 VSOP87 行星理论，该理论不仅能计算各种精密的轨道参数，还可以直接计算出行星的位置，行星位置可以是各种坐标系，包括黄道坐标系。VSOP87 行星理论由 6 张周期项系数表组成，分别是 VSOP87、VSOP87A、VSOP87B、VSOP87C、VSOP87D 和 VSOP87E，其中 VSOP87D 表可以直接计算行星日心黄经（L）、日心黄纬（B）和到太阳的距离（R），此表计算出的结果适用于节气位置判断。

VSOP87D 表包含了三部分数据，分别是 8 大行星的日心黄经周期项系数表（L 表）、日心黄纬周期项系数表（B 表）和行星与太阳距离周期项系数表（R 表）。以地球的数据为例，L 表由 L0 ~ L5 六部分组成，每一部分都包含若干个周期项系数条目，每个周期项系数条目又包含若干个参数，用于计算各种轨道参数和位置参数。计算地球的日心黄经只需要用到其中三个系数。计算所有的周期项系数并不是必须的，有时候减少一些系数比较小的周期项可以减少计算所花费的时间，当然，这会牺牲一点精度。假设计算地球日心黄经的三个系数是 A、B 和 C，则每个周期项的计算表达式是：

$$A * \cos(B + C\tau) \tag{11-11}$$

式(11-11)中的 τ 是儒略千年数，τ 的计算公式如下：

$$\tau = (JDE - 2451545.0) / 365250$$

JDE 是计算轨道参数的时间，单位是儒略日，2451545.0 是公元 2000 年 1 月 1 日 12 时的儒略日数。

以 L0 表的第二个周期项为例，这个周期项数据中与日心黄经计算有关的三个系数分别是 A=3341656.456，B=4.66925680417，C=6283.07584999140，则第二个周期项的计算方法是：3341656.456 * cos(4.66925680417 + 6283.0758499914 *τ)。对 L0 表的各项分别计算后求和可得到 L0 表周期项总和 L0，对 L 表的其他几个部分使用相同的方法计算周期项和，可以得到 L1、L2、

---

① VSOP 行星理论，英文名称是 Secular Variations of the Planetary Orbits。VSOP 的缩写其实源于法文名称：Variations Séculaires des Orbites Planétaires。

L3、L4 和 L5，然后用式(11-12)计算出最终的地球日心黄经，单位是弧度：

$$L = (L0 + L1 * \tau + L2 * \tau^2 + L3 * \tau^3 + L4 * \tau^4 + L5 * \tau^5) / 10^8 \tag{11-12}$$

式(11-12)需要多次计算 τ 的乘方，对于这样的多项式求和，可以利用“霍纳法则”对其进行优化，转化成式(11-13)的形式，虽然形式上繁琐了一点，但是避免了重复计算τ的乘方，非常高效。

用同样的方法对地球日心黄纬的周期项系数表和计算行星和太阳距离的周期项系数表计算求和，可以得到地球日心黄纬 B 和日地距离 R，B 的单位也是弧度，R 的单位则是天文单位（AU）[①]。

VSOP82/87 行星理论中的周期项系数对不同的行星具有不同的精度，对地球来说，在 1900 ~ 2100 年的 200 年跨度期间，计算精度是 0.005"。前面曾说过，对于不需要这么高精度的计算应用时，可以适当减少一些系数比较小的周期项，减少计算量，提高计算速度。Jean Meeus 在他的《天文算法》一书中就给出了一套精简后的 VSOP87D 表的周期项，将计算地球黄经的 L0 表由原来的 559 项精简到 64 项，计算地球黄纬的 B0 表甚至被精简到只有 5 项。从实际效果看，计算精度下降并不多，但是极大地减少了计算量。

使用 VSOP87D 周期项系数表计算得到的是 J2000.0 平黄道和平春分点（mean dynamic ecliptic and equinox）为基准的日心黄经和日心黄纬，其值与标准 FK5 系统[②]略有差别。如果对精度要求很高，可以采用下面的方法将计算得到的日心黄经和日心黄纬转到 FK5 系统。

首先计算 L′，L′的单位是度：

$$L = (((((L5 * \tau + L4) * \tau + L3) * \tau + L2) * \tau + L1) * \tau + L0) / 10^8 \tag{11-13}$$

$$L' = L - 1.397*T - 0.00031*T^2 \tag{11-14}$$

式(11-14)中 T 是儒略世纪数，它与儒略千年数 τ 的计算关系是：$T = 10 * \tau$。计算出 L′之后，就可以利用式(11-15)和式(11-16)，分别计算计算 L 和 B 的修正值 ΔL 和 ΔB：

$$\Delta L = -0.09033 + 0.03916 * (\cos(L') + \sin(L')) * \tan(B) \tag{11-15}$$

$$\Delta B = +0.03916 * (\cos(L') - \sin(L')) \tag{11-16}$$

这里需要注意一点，ΔL 和 ΔB 的单位都是″，是度、分、秒角度单位体系，需要将其转换成

① 天文单位（Astronomical Unit）是一个长度单位，约等于地球与太阳的平均距离。天文单位是天文常数之一，是天文学中测量距离特别是测量太阳系内天体之间的距离的基本单位。地球到太阳的平均距离大约为一个天文单位，约等于 1.496 亿千米。1976 年，国际天文学联会把一天文单位定义为一颗质量可忽略、公转轨道不受干扰而且公转周期为 365.2568983 日（即一高斯年）的粒子与一个质量相等约一个太阳的物体的距离。当前普遍被接受并使用的天文单位的值是 149 597 870 691 ± 30 米（约一亿五千万千米）。

② FK5 是常用的目视星表系统，又称第五基本星表，是在 FK4（第四基本星表）的基础上发展出来的，对 FK4 星表进行了修正，于 1984 年正式启用。它定义了一个以太阳质心为中心，J2000.0 平赤道和春分点为基准的天球平赤道坐标系。近年来国际上又编制了 FK6 星表（第六基本星表），但是还没有正式启用。

弧度单位后再对 L 和 B 进行修正。

CalcSunEclipticLongitudeEC()函数就是使用 VSOP87 行星理论计算行星日心黄经的代码实现，整个计算过程和前文描述一样，首先根据 VSOP87D 表的数据计算出 L0-L5，然后用式(11-13)计算出地球的日心黄经，最后使用式(11-9)将结果转换成太阳的地心黄经。代码如下：

```
double CalcSunEclipticLongitudeEC(double dt)
{

    double L0 = CalcPeriodicTerm(Earth_L0, COUNT_OF(Earth_L0), dt);
    double L1 = CalcPeriodicTerm(Earth_L1, COUNT_OF(Earth_L1), dt);
    double L2 = CalcPeriodicTerm(Earth_L2, COUNT_OF(Earth_L2), dt);
    double L3 = CalcPeriodicTerm(Earth_L3, COUNT_OF(Earth_L3), dt);
    double L4 = CalcPeriodicTerm(Earth_L4, COUNT_OF(Earth_L4), dt);
    double L5 = CalcPeriodicTerm(Earth_L5, COUNT_OF(Earth_L5), dt);

    double L = (((((L5 * dt + L4) * dt + L3) * dt + L2) * dt + L1) * dt + L0) / 100000000.0;

    /*地心黄经= 日心黄经+ 180 度*/
    return L + PI;
}
```

CalcPeriodicTerm()函数使用式(11-11)对 coff 参数指定的周期项系数表进行计算求和计算。采用同样的方法可以计算出太阳的地心黄纬，CalcSunEclipticLatitudeEC()函数首先计算出太阳的日心黄纬，然后用式(11-10)将其转换为地心黄纬。

```
double CalcSunEclipticLatitudeEC(double dt)
{
    double B0 = CalcPeriodicTerm(Earth_B0, COUNT_OF(Earth_B0), dt);
    double B1 = CalcPeriodicTerm(Earth_B1, COUNT_OF(Earth_B1), dt);
    double B2 = CalcPeriodicTerm(Earth_B2, COUNT_OF(Earth_B2), dt);
    double B3 = CalcPeriodicTerm(Earth_B3, COUNT_OF(Earth_B3), dt);
    double B4 = CalcPeriodicTerm(Earth_B4, COUNT_OF(Earth_B4), dt);

    double B = (((((B4 * dt) + B3) * dt + B2) * dt + B1) * dt + B0) / 100000000.0;

    /*地心黄纬= －日心黄纬*/
    return -B;
}
```

计算出地心黄经和地心黄纬之后，就可以使用式(11-15)和式(11-16)的修正计算将其转到 FK5 目视系统，以计算黄经修正量 ΔL 为例，其算法实现如下：

```
double AdjustSunEclipticLongitudeEC(double dt, double longitude, double latitude)
{
    double T = dt * 10; //T是儒略世纪数

    longitude = RadianToDegree(longitude);
    double dbLdash = longitude - 1.397 * T - 0.00031 * T * T;

    // 转换为弧度
    dbLdash *= dbUnitRadian;
```

```
    return (-0.09033 + 0.03916 * (cos(dblLdash) + sin(dblLdash)) * tan(latitude)) / ARC_SEC_PER_RADIAN;
}
```

longitude 参数和 latitude 参数分别是前面计算得到的地心黄经和地心黄纬,返回的结果是地心黄经修正量ΔL。需要注意一点，longitude 参数的单位是弧度，需要转化成度、分、秒单位后才能代入式(11-14)进行计算。

11.2.4 误差修正——章动

经过上述计算转换得到坐标值是理论值，或者说是天体的几何位置，但是 FK5 系统是一个目视系统，也就是说体现的是人眼睛观察效果（光学位置），这就需要根据地球的物理环境、大气环境等信息做进一步的修正，使其和人类从地球上观察星体的观测结果一致。

首先需要进行章动修正。章动是指地球沿自转轴的指向绕黄道极缓慢旋转过程中，由于地球上物质分布不均匀性和月球及其他行星的摄动力造成的轻微抖动。英国天文学家詹姆斯·布拉德利(1693—1762)最早发现了章动，章动可以沿着黄道分解为水平分量和垂直分量，黄道上的水平分量记为Δψ，称为黄经章动，它影响了天球上所有天体的经度。黄道上的垂直分量记为Δε，称为交角章动，它影响了黄赤交角。目前编制天文年历所依据的章动理论是伍拉德在 1953 年建立的，它是以刚体地球模型为基础的。1977 年，国际天文联合会的一个专家小组建议采用非刚体地球模型——莫洛坚斯基 II 模型代替刚体地球模型计算章动，1979 年的国际天文学联合会第十七届大会正式通过了这一建议，并决定于 1984 年正式实施。

地球章动主要是月球运动引起的，也具有一定的周期性，可以描述为一些周期项的和，主要项的周期是 6798.4 日（18.6 年），但其他项是一些短周期项（小于 10 天）。本文采用的计算方法取自国际天文联合会的 IAU1980 章动理论，周期项系数数据来源于《天文算法》一书第 21 章的表 21-A，该表忽略了 IAU1980 章动理论中系数小于 0.0003″的周期项，因此只有 63 项。每个周期项包括计算黄经章动（Δψ）的正弦系数（相位内项系数）、计算交角章动的（Δε）余弦系数（相位外项系数）以及计算辐角的 5 个基本角距（M、M'、D、F、Ω）的线性组合系数。5 个基本角距的计算公式分别如下所示。

平距角（日月对地心的角距离）计算公式：

$$D = 297.85036 + 455267.111480 * T - 0.0019142 * T^2 + T^3 / 189474 \quad (11\text{-}17)$$

太阳（地球）平近点角计算公式：

$$M = 357.52772 + 35999.050340 * T - 0.0001603 * T^2 - T^3 / 300000 \quad (11\text{-}18)$$

月球平近点角计算公式：

$$M' = 134.96298 + 477198.867398 * T + 0.0086972 * T^2 + T^3 / 56250 \quad (11\text{-}19)$$

月球纬度参数计算公式：

$$F = 93.27191 + 483202.017538 * T - 0.0036825 * T^2 + T^3 / 327270 \tag{11-20}$$

黄道与月球平轨道升交点黄经：

$$\Omega = 125.04452 - 1934.136261 * T - 0.0020708 * T^2 + T^3 / 450000 \tag{11-21}$$

以上各式中的 T 是儒略世纪数，计算出来的 5 个基本角距的单位都是度，在计算正弦或余弦时要转换为弧度单位。计算每一个周期项的黄经章动过程是这样的，首先用式(11-17)至式(11-20)计算出 5 个基本角距，然后将计算出来的值与对应的 5 个基本角距系数组合，计算出辐角 θ。以本节使用的章动周期项系数表中的第七项为例，5 个基本角距对应的系数分别是 1、0、－2、2 和 2，则辐角 θ 的值就是：

$$\theta = -2D + M + 2F + 2\Omega$$

计算出辐角后就可以使用式(11-22)计算周期项的值：

$$S = (S1 + S2 * T) * \sin(\theta) \tag{11-22}$$

使用式(11-22)计算出各周期项的值后累加求和就可得到黄经章动，注意，黄经章动的单位是 0.0001″，对地心黄经进行修正时需要转换成弧度单位。交角章动的计算方法与黄经章动的计算类似，辐角 θ 的值是一样的，只是计算章动使用的是余弦系数：

$$C = (C1 + C2 * T) * \cos(\theta) \tag{11-23}$$

CalcEarthLongitudeNutation()函数计算黄经章动，计算交角章动的算法实现与之类似。GetEarthNutationParameter()辅助函数用于计算 5 个基本角距，就是式(11-17)至式(11-20)的具体实现。最终计算的结果已经转换成弧度单位，可以直接对之前已经转换到 FK5 系统的地心黄经进行章动修正。

```
double CalcEarthLongitudeNutation(double dt)
{
    double T = dt * 10;
    double D,M,Mp,F,Omega;

GetEarthNutationParameter(dt, &D, &M, &Mp, &F, &Omega);

    double resulte = 0.0 ;
    for(int i = 0; i < COUNT_OF(nutation); i++)
    {
        double sita = nutation[i].D * D + nutation[i].M * M + nutation[i].Mp * Mp + nutation[i].F *
             F + nutation[i].omega * Omega;
        sita = DegreeToRadian(sita);

        resulte += (nutation[i].sine1 + nutation[i].sine2 * T ) * sin(sita);
    }

    /*先乘以章动表的系数 0.001"，然后换算成弧度的单位*/
    return resulte * 0.0001 / ARC_SEC_PER_RADIAN;
}
```

11.2.5 误差修正——光行差

除了章动修正，对于目测系统来说，还要进行光行差修正。光行差是指在同一瞬间，运动中的观察者所观测到的天体视方向与静止的观测者所观测到天体的真方向之差。造成光行差的原因有两个，一个是光的有限速度，另一个是观察者的运动。在地球上的天文观测者因和地球一起运动（自转 + 公转），他所看到的星光方向与假设地球不动时看到的方向不一样。以太阳为例，光线从太阳传到地球需要约 8 分钟的时间，在这 8 分钟多的时间中，地球沿着公转轨道移动了一段距离，人们根据现在的观察认定太阳在那个视位置，事实上那是 8 分钟前太阳的位置。在精确的天文计算中，需要考虑这种光行差引起的视位置差异，在计算太阳的地心视黄经时，要对其进行光行差修正。地球上的观测者可能会遇到几种光行差，分别是因地球公转引起的周年光行差，因地球自转引起的周日光行差，还有因太阳系或银河系运动形成的长期光行差等，对于从地球上观察太阳这种情况，只需要考虑周年光行差和周日光行差。因太阳公转速度比较快，周年光行差最大可达到 20.5 角秒，在计算太阳视黄经时需要考虑修正。地球自转速度比较慢，周日光行差最大约为零点几个角秒，因此计算太阳视黄经时忽略周日光行差。

下面是一个粗略计算太阳地心黄经光行差修正量的公式，其中 R 是地球和太阳的距离：

$$\mathrm{AC} = -20''.4898 / R \tag{11-24}$$

分子 20.4898 并不是一个常数，但是其值的变化非常缓慢，在 0 年是 20″.4893，在 4000 年是 20″.4904。前面提到过，太阳到地球的距离 R 可以用 VSOP87D 表的 $R0 \sim R5$ 周期项计算出来，R 的单位是“天文单位”（AU），和计算太阳地心黄经和地心黄纬类似，太阳到地球的距离可以这样算出来：

```
double CalcSunEarthRadius(double dt)
{
    double R0 = CalcPeriodicTerm(Earth_R0, COUNT_OF(Earth_R0), dt);
    double R1 = CalcPeriodicTerm(Earth_R1, COUNT_OF(Earth_R1), dt);
    double R2 = CalcPeriodicTerm(Earth_R2, COUNT_OF(Earth_R2), dt);
    double R3 = CalcPeriodicTerm(Earth_R3, COUNT_OF(Earth_R3), dt);
    double R4 = CalcPeriodicTerm(Earth_R4, COUNT_OF(Earth_R4), dt);

    double R = (((((R4 * dt) + R3) * dt + R2) * dt + R1) * dt + R0) / 100000000.0;

    return R;
}
```

计算出太阳到地球的距离之后，就可以使用式(11-24)计算光行差修正量。式(11-24)计算出的结果是度、分、秒单位（角秒），需要转换成弧度单位，这个转换在 AdjustSunEclipticLongitudeAberration()函数中体现。

```
double AdjustSunEclipticLongitudeAberration(double dt)
{
    double dtmp = -20.4898 / CalcSunEarthRadius(dt);
    return dtmp / ARC_SEC_PER_RADIAN;
}
```

11.2.6　用牛顿迭代法计算二十四节气

由 VSOP87 理论计算出来的几何位置黄经，经过坐标转换，章动修正和光行差修正后，就可以得到比较准确的太阳地心视黄经，GetSunEclipticLongitudeEC()函数就是整个过程的体现，参数 dt 是儒略千年数，它与儒略日的计算关系在 11.2.3 节已经给出。

```
double GetSunEclipticLongitudeEC(double dt)
{
    // 计算太阳的地心黄经
    double longitude = CalcSunEclipticLongitudeEC(dt);

    // 计算太阳的地心黄纬
    double latitude = CalcSunEclipticLatitudeEC(dt);

    //校正经度
    longitude += AdjustSunEclipticLongitudeEC(dt, longitude, latitude);

    //天体章动修正
    longitude += CalcEarthLongitudeNutation(dt);

    /*太阳地心黄经光行差修正*/
    longitude += AdjustSunEclipticLongitudeAberration(dt);

    return longitude;
}
```

到现在为止，我们已经知道如何使用 VSOP82/87 理论计算以儒略日为单位的任意时刻的太阳地心视黄经，但是这和实际历法计算需求还不一致，历法计算需要根据太阳地心视黄经反求出此时的时间。VSOP82/87 理论没有提供反向计算的方法，但是可以采用根据时间正向计算太阳视黄经，配合误差修正进行迭代计算的方法，使正向计算出来的结果向已知结果收敛，当达到一定的迭代次数或计算结果与已知结果误差满足精度要求时，停止迭代，此时的正向输入时间就是所求的时间。地球公转轨道是近似椭圆轨道，轨道方程不具备单调性，但是在某个节气附近的一小段时间区间中，轨道方程具有单调性，这个是本节迭代算法的基础。

实际上，我们要做的事情就是求解方程的根，但是我们面临的这个方程没有解析表达式，更不用说求根公式了。本书第 13 章介绍了几种迭代法求解非线性方程的方法，都适用于我们现在面临的问题。牛顿迭代法具有收敛速度快、稳定的特点，所以我们选择使用牛顿迭代法求解这个问题。使用牛顿迭代法首先要定义函数 $f(x)$。我们观察 GetSunEclipticLongitudeEC()函数，参数 dt 是一个与时间有关的变量，返回值是一个角度值（太阳的地心视黄经），如果将 dt 视为自变量，返回值 angle 视为结果，则 $f(x)$可定义为：

```
f(x) = GetSunEclipticLongitudeEC(x) - angle
```

angle 是节气对应的地心黄经角度，对每个节气来说，angle 是个常量。定义了 $f(x)$，就可以写出牛顿迭代关系：

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

确定了方程 $f(x)$，剩下的问题就是求导函数 $f'(x)$。严格的求解，应该根据 GetSunEclipticLongitudeEC()函数，以儒略千年数 dt 为自变量，按照函数求导的规则求出导函数。因为 GetSunEclipticLongitudeEC()函数内部是调用其他函数，因此可以理解为是一个多个函数组合的复合函数，类似 $f(x) = g(x) + h(x, k(x)) + p(x)$这样的形式，可以按照求导规则逐步对其求导得到导函数。但是我不打算这么做，因为有更简单的方法。第 13 章介绍牛顿迭代法一节介绍了求一阶导数的近似公式，其实求导函数的目的就是为了得到某一点的导数，如果有近似公式可以直接得到这一点的导数，就不用费劲求导函数了。有关近似公式的说明可参考第 13 章对近似公式的描述，这里就直接用了：

$$f'(x_0) = (f(x_0 + 0.000005) - f(x_0 - 0.000005)) / 0.00001 \tag{11-25}$$

牛顿迭代法在进行迭代求解时，需要指定一个迭代初始值，初始值的选择越接近问题的解，迭代收敛的速度就越快。当我们求一个节气的准确时间时，我们希望从一个比较接近准确时间的时间开始迭代。根据节气日期的规律，每个月的节气时间比较固定，最多相差一两天，考虑到几千年后岁差的影响，这个估算范围还可以再放宽一点。比如，对于月内的第一个节气，可以将时间范围估算为 4 日到 9 日，对于月内的第二个节气，可以将时间范围估算为 16 日到 24 日，保证迭代范围内有解。为此，我们取第一个节气时间为每月的 6 日，第二个节气时间为每月的 20 日。根据节气的规律，我们知道节气和月份存在对应关系，因此根据节气对应的太阳地心黄经角度，可以反推出月份。结合指定的年份、根据节气反推出来的月份和估计的日期，就可以计算出儒略日，这个就是迭代的初始值。估算迭代初始值的算法就体现在 GetInitialEstimateSolarTerms()函数内，angle 参数就是节气对应的太阳地心视黄经，这个角度值和节气是固定的对应关系。

```
double GetInitialEstimateSolarTerms(int year, int angle)
{
    int STMonth = int(ceil(double((angle + 90.0) / 30.0)));
    STMonth = STMonth > 12 ? STMonth - 12 : STMonth;

    /* 每月第一个节气发生日期基本都-9 日之间，第二个节气的发生日期都在 – 日之间*/
    if((angle % 15 == 0) && (angle % 30 != 0))
    {
        return CalculateJulianDay(year, STMonth, 6, 12, 0, 0.00);
    }
    else
    {
        return CalculateJulianDay(year, STMonth, 20, 12, 0, 0.00);
    }
}
```

有了求导数的近似公式，有了迭代初始值，就可以根据牛顿迭代关系写出迭代求解的算法，正如你在 CalculateSolarTerms()函数中看到的那样，非常简单。唯一需要特殊说明的是，由于角度的 360 度圆周性，当在太阳黄经 0 度附近逼近时，迭代可能是从(345, 360]和[0, 15)两个方向上向 0 逼近，此时需要将(345, 360]区间修正为(–15, 0]，使得逼近区间边界的选取能够正常进行。经过验证，牛顿迭代法具有非常好的收敛效果，一般只需 3 次迭代就可以得到满足精度的结果。

11

```
double CalculateSolarTerms(int year, int angle)
```

```
{
    double JD0, JD1,stDegree,stDegreep;

    JD1 = GetInitialEstimateSolarTerms(year, angle);
    do
    {
        JD0 = JD1;
        stDegree = GetSunEclipticLongitudeEC(JD0);
        /*
            对黄经度迭代逼近时，由于角度度圆周性，估算黄经值可能在(345,360]和[0,15)两个区间，如果值
            落入前一个区间，需要进行修正
        */
        stDegree = ((angle == 0) && (stDegree > 345.0)) ? stDegree - 360.0 : stDegree;
        stDegreep = (GetSunEclipticLongitudeEC(JD0 + 0.000005)
                    - GetSunEclipticLongitudeEC(JD0 - 0.000005)) / 0.00001;
        JD1 = JD0 - (stDegree - angle) / stDegreep;
    }while((fabs(JD1 - JD0) > 0.0000001));

    return JD1;
}
```

至此，我们就有了完整的计算节气发生时间的方法，输入年份和节气对应的太阳黄经度数，即可求的该节气发生的精确时间。最后说明一下，以上算法中讨论的时间都是力学时时间（TD）[①]，与国际协调时[②]（UTC）以及各个时区的本地时间都有不同，需要将计算出的结果转换成国际协调时，然后再调整到适当的时区，比如中国的中原地区就是东八区标准时（UTC+8）。应用本节的算法计算出 2012 年各个节气的时间如下（已经转换为东八区标准时），与紫金山天文台发布的《2012 中国天文年历》中发布的时间在分钟级别上完全吻合（此年历只精确到分钟）：

```
2012-01-06, 06:43:54.28    小寒
2012-01-21, 00:09:49.08    大寒
2012-02-04, 18:22:22.53    立春
……
```

11.3　农历朔日（新月）的天文学计算

除了公历的一些算法，本章还要介绍中国农历的天文计算，农历是一种极具中国特色的日月结合的历法。中国农历的朔望月是农历历法的基础，而朔望月又是严格以日月合朔发生的那一天

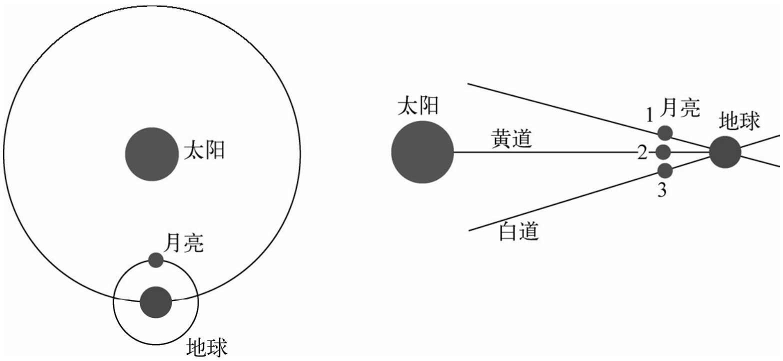
① 力学时，全称是“牛顿力学时”，也称作“历书时”。它描述天体运动的动力学方程中作为时间自变量所体现的时间，或天体历表中应用的时间，是由天体力学的定律确定的均匀时间。力学时的初始历元取为 1900 年初附近，太阳几何平黄经为 279°41′48″.04 的瞬间，秒长定义为 1900.0 年回归年长度的 1 / 31556925.9747。1958 年国际天文学联合会决议决定：自 1960 年开始用力学时代替世界时作为基本的时间计量系统，规定天文年历中太阳系天体的位置都按力学时推算。力学时与世界时之差由观测太阳系天体（主要是月球）定出，因此力学时的测定精度较低，1967 年起被原子时代替作为基本时间计量系统。

② 国际协调时又称世界时，是以本初子午线的平子夜起算的平太阳时，又称格林威治时间。世界各地地方时与世界时之差等于该地的地理经度。世界时 1960 年以前曾作为基本时间计量系统被广泛应用。由于地球自转速度变化的影响，它不是一种均匀的时间系统。后来世界时先后被历书时和原子时所取代。

作为月首，因此日月合朔时间的计算是制定农历历法的关键。本节将介绍 ELP-2000/82 月球运行理论，以及如何用 ELP-2000/82 月球运行理论计算日月合朔时间。

11.3.1　日月合朔的天文学定义

要计算日月合朔时间，首先要对日月合朔这一天文现象进行数学定义。朔望月是在地球上观察到的月相周期，平均长度约等于 29.53059 日，而恒星月（天文月）是月亮绕地球公转一周的时间，长度约 27.32166 日。月相周期长度比恒星月长大约两天，这是因为在月球绕地球旋转一周的同时，地球还带着它绕太阳旋转了一定的角度的缘故，所以月相周期不仅与月球运行有关，还和太阳运行有关。日月合朔的时候，太阳、月亮和地球三者接近一条直线，月亮未被照亮的一面对着地球，因此地球上看不到月亮，此时又被称为新月。图 11-3a 就是日月合朔天文现象的示意图。



(a)　　(b)

图 11-3　日月天文现象示意图

月亮绕太阳公转的白道面和地球绕太阳公转的黄道面存在一个最大约 5° 的夹角，因此大多数情况下，日月合朔时都不是严格在同一条直线上，不过也会发生在同一直线的情况，此时就会发生日食。图 11-3b 显示了日月合朔时侧切面上月亮的三种可能的位置情况，当月亮处在位置 2 时就会发生日食。由图 11-3 可知，日月合朔的数学定义就是太阳和月亮的地心视黄经差为 0 的时刻。

11.3.2　ELP-2000/82 月球理论

要计算日月合朔，需要知道太阳地心视黄经和月亮地心视黄经的计算方法。本章 11.2 节已经介绍了如何用 VSOP82/87 行星理论计算太阳的地心视黄经，本节将介绍如何用 ELP-2000/82 月球理论计算月亮的地心视黄经。有了太阳地心视黄经和月亮地心视黄经的计算方法，就可以反向推算它们相等的时间，这个时间就是日月合朔的时间。

ELP-2000/82 月球理论是 M. Chapront-Touze 和 J. Chapront 在 1983 年提出的一个月球位置的

半解析理论。和其他半解析理论一样，ELP-2000/82 理论也包含一套计算方法和相应的迭代周期项。这套理论共包含 37862 个周期项，其中 20560 个用于计算月球经度，7684 个用于计算月球纬度，9618 个用于计算地月距离。但是这些周期项中有很多都是非常小的值，例如一些计算经纬度的项对结果的增益只有 0.00001 角秒,还有一些地月距离周期项对距离结果的增益只有 0.02 米，对于精度不高的历法计算，完全可以忽略。

有很多基于 ELP-2000/82 月球理论的改进或简化理论，《天文算法》一书的第四十五章就介绍了一种改进算法，其周期项参数都是从 ELP-2000/82 理论的周期项参数转换来的，忽略了小的周期项。使用该方法计算的月球黄经精度只有 10″，月亮黄纬精度只有 4″，但是只用计算 60 个周期项，速度很快，本节就采用这种修改过的 ELP-2000/82 理论计算月亮的地心视黄经。这种计算方法的周期项分三部分，分别用来计算月球黄经、月球黄纬和地月距离，三部分的周期项的内容一样，由四个计算辐角的系数和一个正弦（或余弦）振幅组成。计算月球黄经和月球黄纬使用正弦表达式求和：

$$A * \sin(\theta) \tag{11-26}$$

计算地月距离用余弦表达式求和：

$$A * \cos(\theta) \tag{11-27}$$

其中辐角 θ 的计算公式是：

$$\theta = a * D + b * M + c * M' + d * F \tag{11-28}$$

式(11-28)中的四个辐角系数 a、b、c 和 d 由每个迭代周期项给出，日月距角 D、太阳平近地角 M、月亮平近地角 M'以及月球生交点平角距 F 则分别由式(11-29)至式(11-32)进行计算：

$$D = 297.8502042 + 445267.1115168*\mathrm{T} - 0.0016300*\mathrm{T}^2 + \mathrm{T}^3 / 545868 - \mathrm{T}^4 / 113065000 \tag{11-29}$$

$$M = 357.5291092 + 35999.0502909*\mathrm{T} - 0.0001536*\mathrm{T}^2 + \mathrm{T}^3 / 24490000 \tag{11-30}$$

$$M' = 134.9634114 + 477198.8676313*\mathrm{T} + 0.0089970*\mathrm{T}^2 + \mathrm{T}^3 / 69699 - \mathrm{T}^4 / 14712000 \tag{11-31}$$

$$F = 93.2720993 + 483202.0175273 * \mathrm{T} - 0.0034029*\mathrm{T}^2 - \mathrm{T}^3 / 3526000 + \mathrm{T}^4 / 863310000 \tag{11-32}$$

以上各式计算结果的单位是度，其中的 T 是儒略世纪数，它与儒略千年数的关系及计算方法已经在 11.2.3 节给出。以计算月球黄经的周期项第二项的计算为例，第二项数据分别是：辐角系数 $a = 2$，$b = 0$，$c = -1$，$d = 0$，振幅 $A = 1274027$，黄经计算用正弦表达式，则 I_2 的计算如下所示：

$$I_2 = 1274027 * \sin(2D - M')$$

11.3.3 误差修正——地球轨道离心率修正

在套用式(11-26)和式(11-27)计算出月球黄经周期项和月球黄纬周期项的时候，需要注意对包含了太阳平近地角 M 的项进行修正，因为 M 的值与地球公转轨道的离心率有关，因为离心率是个与时间有关的变量，导致振幅 A 实际上是个变量，需要根据时间进行修正。月球黄经周期项的修正方法是：如果辐角中包含了 M 或 $-M$ 时，需要乘以系数 E 修正；如果辐角中包含了 $2M$ 或 $-2M$，则需要乘以系数 E 的平方进行修正。系数 E 的计算表达式如下：

$$E = 1 - 0.002516 * \mathrm{T} - 0.0000074 * \mathrm{T}^2 \tag{11-33}$$

这个修正可以在计算周期项的时候直接进行，以计算月球黄经周期项的算法实现函数为例，CalcMoonECLongitudePeriodic()函数在计算每一项周期项时，直接乘以 pow(E,fabs(Moon_longitude[i].M))进行修正，使用 pow()函数并不是一个高效的方法，此处使用 pow()函数仅仅是为了利用了 E 的 0 次幂结果是 1 的数学特性，省去几行代码，大家可自行体会。使用式(11-28)计算出的辐角 sita 是角度单位，需要转换成弧度单位才能调用 sin()函数，这个在代码中都有体现。

```
double CalcMoonECLongitudePeriodic(double D, double M, double Mp, double F, double E)
{
    double EI = 0.0 ;

    for(int i = 0; i < COUNT_OF(Moon_longitude); i++)
    {
        double sita = Moon_longitude[i].D * D + Moon_longitude[i].M * M + Moon_longitude[i].Mp * Mp
            + Moon_longitude[i].F * F;
        sita = DegreeToRadian(sita);
        EI += (Moon_longitude[i].eiA * sin(sita) * pow(E, fabs(Moon_longitude[i].M)));
    }

    return EI;
}
```

调用 CalcMoonECLongitudePeriodic()函数得到地球轨道离心率修正后的月球黄经周期项之和 ΣI，计算月球黄纬同样需要根据 M 对周期项结果进行修正，修正的方法和对月球黄经的修正方法相同，因此可以使用相同的方法得到修正地球轨道离心率之后的月球黄纬周期项之和 Σb。一般来说，计算地月距离的目的是为了计算月亮光行差，但是因为地月距离较小，从地球观察月亮产生的光行差也很小，相对于本章介绍的历法的算法的精度（月球黄经精度 10″，月亮黄纬精度 4″）来说，可以忽略光行差修正，因此就不用计算地月距离。

11.3.4 误差修正——黄经摄动

金星轨道距离地月系轨道比较近，金星的运行会对月球的运行产生摄动影响。木星的轨道虽然距离地月系轨道远一点，但是质量大，因此木星的运行同样会对月球的运行产生摄动影响。与此同时，因为地球不是一个规则的刚性球体，因此地球形状的不规则性也会对月球的运行产生影响，这些影响会导致月球运行时产生黄经摄动，因此需要对计算出的月球黄经周期项和 ΣI 与月

球黄纬周期项和 Σb 进行摄动修正，修正的方法如下：

$$\Sigma I \mathrel{+}= +3958*\sin(A_1) + 1962*\sin(L' - F) + 318*\sin(A_2) \tag{11-34}$$

$$\Sigma b \mathrel{+}= -2235*\sin(L') + 382*\sin(A_3) + 175*\sin(A_1 - F) + 175*\sin(A_1 + F) + 127*\sin(L' - M') - 115*\sin(L' + M') \tag{11-35}$$

其中 M'和 F 分别由式(11–31)和式(11–32)计算得到，L'是月球平黄经，计算方法是：

$$L' = 218.3164591 + 481267.88134236*T - 0.0013268*T^2 + T^3 / 538841 - T^4 / 65194000 \tag{11-36}$$

A_1 是与金星相关的摄动角修正量，A_2 是与木星相关的摄动角修正量，L'和 A_3 是与地球扁率摄动相关的摄动角修正量，这三个修正量的计算方法如下：

$$A_1 = 119.75 + 131.849 * T \tag{11-37}$$

$$A_2 = 53.09 + 479264.290 * T \tag{11-38}$$

$$A_3 = 313.45 + 481266.484 * T \tag{11-39}$$

月球地心黄经摄动的修正使用式(11-34)给出的方法计算，实现算法如下：

```
double CalcMoonLongitudePerturbation(double dt, double Lp, double F)
{
    double T = dt; /*T 是从 J2000 起算的儒略世纪数*/
    double A1 = 119.75 + 131.849 * T;
    double A2 = 53.09 + 479264.290 * T;

    double result = 3958.0 * sin(DegreeToRadian(A1));
    result += (1962.0 * sin(DegreeToRadian(Lp - F)));
    result += (318.0 * sin(DegreeToRadian(A2)));

    return result;
}
```

CalcMoonLongitudePerturbation()函数计算出的结果是摄动修正量，最后需要将这个结果与前面计算出来的月球黄经周期项和 ΣI 进行叠加，得到修正后的结果。再次提醒一下，根据式(11-37)和式(11-38)计算出来的 A_1 和 A_2 单位是度，需要转换为弧度才能调用 sin()函数进行计算。

11.3.5 月球地心视黄经和最后的修正——地球章动

完成所有的修正之后，需要对 ΣI 和 Σb 进行最后的计算，得到月球地心视黄经 λ 和月球地心视黄纬 β，这个最后的计算公式如下所示：

$$\lambda = L' + \Sigma I / 1000000.0 \tag{11-40}$$

$$\beta = \Sigma b / 1000000.0 \tag{11-41}$$

L'是用式(11-36)计算出来的月球平黄经。对 ΣI 和 Σb 除以 1000000.0 的原因是周期项系数中振幅 A 的单位是 0.000001 度。最终得到的月球地心视黄经 λ 和月球地心视黄纬 β 的单位是度。

到此并没有结束，还有一项修正需要考虑。前面已经提到过，地球不是圆球刚体，其不规则形状会对在地球上的目视观察系统产生影响，那就是地球的章动。11.2.4 节已经介绍过地球章动对太阳地心视黄经的影响和修正算法，该算法对月球的地心视黄经同样适用。11.2.4 节已经给出了地球章动对黄经的修正的实现函数 CalcEarthLongitudeNutation()，将这个结果叠加到之前计算出的月球地心视黄经 λ 上即可完成章动修正。对于月球地心黄纬，同样要适用交角章动进行修正，但是计算如月合朔只需要计算月球地心黄经即可，对月球地心黄纬的修正算法此处就不列出了，读者可在本书的配套代码中找到它们。

完整的周期项计算、修正并最后转换出月球地心视黄经结果的算法实现就是函数 GetMoonEclipticLongitudeEC()。参数 dbJD 是指定时间的儒略日，返回结果是月球地心视黄经，单位是度。

```
double GetMoonEclipticLongitudeEC(double dbJD)
{
    double Lp,D,M,Mp,F,E;
    double dt = (dbJD - JD2000) / 36525.0; /*儒略世纪数*/

    GetMoonEclipticParameter(dt, &Lp, &D, &M, &Mp, &F, &E);

    /*计算月球地心黄经周期项*/
    double EI = CalcMoonECLongitudePeriodic(D, M, Mp, F, E);

    /*修正金星、木星以及地球扁率摄动*/
    EI += CalcMoonLongitudePerturbation(dt, Lp, F);

    /*计算月球地心视黄经*/
    double longitude = Lp + EI / 1000000.0;

    /*计算天体章动干扰*/
    longitude += RadianToDegree(CalcEarthLongitudeNutation(dt / 10.0));

    return longitude;
}
```

11.3.6 用牛顿迭代法计算日月合朔

至此，我们有了用半解析理论计算月球地心视黄经的算法，但是和节气的计算一样，历法的计算需要根据月球的地心视黄经反推对应的时间，这就像解方程一样，我们仍然需要一个反向计算的结果。为此，我们再次选择牛顿迭代法。使用牛顿迭代法，需要指定函数 $f(x)$。日月合朔的天文定义是太阳地心视黄经和月球地心视黄经相等的那一刻，也就是它们的差值是 0 的那一刻，于是我们这样确定 $f(x)$：

```
f(x) = GetSunEclipticLongitudeEC(x) - GetMoonEclipticLongitudeEC(x)
```

我们需要求解 $f(x) = 0$ 的时候的解 x，这个 x 其实就是对应的儒略日时间。

牛顿迭代关系和一阶导数近似公式请参看 11.2.6 节的方法，这里不再赘述。这个算法需要注意的地方就是角度的 360 度周期性，在 0 度（360 度）附近要特殊处理，处理的方法就是按照 360

圆整，避免从角度理解应该是很接近的两个值，相减的结果却是一个很大的值。具体的算法实现请参考 CalculateMoonShuoJD()函数。入参 tdJD 是迭代初始值，这个初始值可以根据朔望月的平均长度 29.53059 进行适当的估算。

```
double CalculateMoonShuoJD(double tdJD)
{
    double JD0, JD1,stDegree,stDegreep;

    JD1 = tdJD;
    do
    {
        JD0 = JD1;
        double moonLongitude = GetMoonEclipticLongitudeEC(JD0);
        double sunLongitude = GetSunEclipticLongitudeEC(JD0);
        if((moonLongitude > 330.0) && (sunLongitude < 30.0))
        {
            sunLongitude = 360.0 + sunLongitude;
        }
        if((sunLongitude > 330.0) && (moonLongitude < 30.0))
        {
            moonLongitude = 60.0 + moonLongitude;
        }

        stDegree = moonLongitude - sunLongitude;
        stDegree = Mod360Degree(stDegree);
        stDegreep = (GetMoonEclipticLongitudeEC (JD0 + 0.000005) - GetSunEclipticLongitudeEC (JD0 +
            0.000005) - GetMoonEclipticLongitudeEC (JD0 - 0.000005) + GetSunEclipticLongitudeEC
            (JD0 - 0.000005)) / 0.00001;
        JD1 = JD0 - stDegree / stDegreep;
    }while((fabs(JD1 - JD0) > 0.00000001));

    return JD1;
}
```

检验一下我们的算法吧，我们用 CalculateMoonShuoJD()函数计算了农历 2015 年的前三个朔日，分别是：

```
2015-02-19, 07:47:17.38      春节
2015-03-20, 17:36:12.32      二月初一
2015-04-19, 02:56:57.98      三月初一
```

大家可以和 2015 年的日历对一下，看看准不准。

11.4 农历的生成算法

世界各国的日历都是以天为最小单位，但是关于年和月的算法却各不相同，大致可以分为以下三类。

- ❑ 阳历，以天文年作为日历的主要周期，例如中国公历（格里历）。
- ❑ 阴历，以天文月作为日历的主要周期，例如伊斯兰历。

❑ 阴阳历，以天文年和天文月作为日历的主要周期，例如中国农历。

我国古人很早就开始关注天象，定昼夜交替为“日”，月轮盈亏为“月”，寒暑交替为“年”，在总结日月变化规律的基础上制定了兼有阴历月和阳历年性质的历法，称为中国农历。本节将介绍中国农历的历法规则、天干地支的计算方法、二十四节气与中国农历的关系，以及在知道节气和日月合朔的精确时间的情况下推算中国农历年历的方法。

11.4.1 中国农历的起源与历法规则

在介绍中国农历的历法之前，必须要先介绍一下中国古代的纪年方法。中国古代用天干地支纪年，严格来讲，天干地支纪年以及十二属相并不是中国农历历法的一部分，但是在中国历史上直到今天，天干地支以及十二属相一直都是中国农历纪年关系密切的一部分，因此这里先介绍一下天干地支纪年法以及十二属相。

1. 天干地支与十二生肖

中国古代纪年不用数字，而是采用天干地支组合。天干有十个，分别是：甲、乙、丙、丁、戊、己、庚、辛、壬、癸；地支有十二个，分别是：子、丑、寅、卯、辰、巳、午、未、申、酉、戌、亥。使用时天干地支各取一字，天干在前，地支在后，组合成干支，例如甲子、乙丑、丙寅等，依次轮回可形成六十种组合，以这些天干地支组合纪年，每六十年一个轮回，称为一个甲子。实际上中国古代纪月、纪日以及纪时辰都采用干支方法，这些干支组合起来就是我们熟悉的生辰八字。

十二属相又称“十二生肖”，由十一种源自自然界的动物：鼠、牛、虎、兔、蛇、马、羊、猴、鸡、狗、猪以及传说中的龙组成，用于纪年时，按顺序和十二地支组合成子鼠、丑牛、寅虎、卯兔、辰龙、巳蛇、午马、未羊、申猴、酉鸡、戌狗和亥猪。天干地支以及十二生肖常组合起来描述农历年，比如公历 2011 年就是农历辛卯兔年，2012 年是壬辰龙年等。

计算某一年的天干地支，有很多经验公式，如果知道某一年的天干地支，也可以直接推算其他年份的天干地支。举个例子，如果知道 2000 年是庚辰龙年，则 2012 年的干支可以这样推算：（2012 – 2000）% 10=2，2012 年的天干就是从庚开始向后推 2 个天干，即壬。2012 年的地支可以这样推算：（2012 – 2000）% 12 = 0，2012 年的地支仍然是辰，因此 2012 年的天干地支就是壬辰，十二生肖龙年。对于 2000 年以前的年份，计算出年份差后只要将天干和地支向前推算即可。例如 1995 年的干支可以这样计算：（2000 – 1995）%10 = 5，（2000 – 1995）%12 = 5，庚向前推算 5 即是乙，辰向前推算 5 即是亥，因此 1995 年的干支就是乙亥，十二生肖猪年。这个干支推算算法的实现如下：

```
void CalculateStemsBranches(int year, int *stems, int *branches)
{
    int sc = year - 2000;
    *stems = (7 + sc) % 10;
    *branches = (5 + sc) % 12;
```

```
    if(*stems < 0)
        *stems += 10;
    if(*branches < 0)
        *branches += 12;
}
```

定义好干支和十二生肖的名称数组，就可以实现简单的干支纪年查询功能：

```
TCHAR *nameOfStems[HEAVENLY_STEMS] = { _T("甲"),_T("乙"),_T("丙"),_T("丁"),_T("戊"),_T("己"),
_T("庚"),_T("辛"),_T("壬"),_T("癸") };
TCHAR *nameOfBranches[EARTHLY_BRANCHES] = { _T("子"),_T("丑"),_T("寅"),_T("卯"),_T("辰"),
_T("巳"),_T("午"),_T("未"),_T("申"),_T("酉"),_T("戌"),_T("亥") };
TCHAR *nameOfShengXiao[CHINESE_SHENGXIAO] = { _T("鼠"),_T("牛"),_T("虎"),_T("兔"),_T("龙"),
_T("蛇"),_T("马"),_T("羊"),_T("猴"),_T("鸡"),_T("狗"),_T("猪") };

int stems,branchs;
CalculateStemsBranches(2008, &stems, &branchs);
text.Format(_T("农历【%s%s】%s 年"), m_curMonth, nameOfStems[stems - 1], nameOfBranches[branchs - 1],
nameOfShengXiao[branchs - 1]);
2008 年是农历【戊子】鼠年
```

2. 农历闰月与二十四节气的关系

中国农历是以月亮运行周期为基础，结合太阳运行规律（二十四节气）制定的历法，农历月的定义规则就是中国农历历法的关键，因此要了解中国农历的历法规则，就必须知道如何定义月，如何设置闰月？中国农历的一年有十二个月或十三个月，但是正统的叫法只有十二个月，分别是正月、二月、三月、四月、五月、六月、七月、八月、九月、十月、冬月和腊月（注意，正统的中国农历是没有十一月和十二月的，如果你用的历法软件有显示农历十一月和农历十二月，就说明非常不专业）。中国民间常用“十冬腊月天”来形容寒冷的天气，其实指的就是十月、十一月和十二月这三个最冷的月份。一年有十三个月的情况是因为有闰月，多出来的这个闰月没有月名，只是跟在某个月后面，称为闰某月。比如公历 2009 年对应的农历乙丑年，就是闰五月，于是这一年可以过两个端午节。

中国农历为什么会有闰月？其实中国农历置闰月是为了协调回归年和农历年的矛盾。前面提到过，中国农历是一种阴阳历，农历的月分大月和小月，大月一个月是 30 天，小月一个月是 29 天。中国农历把日月合朔（太阳和月亮的黄经相同，但是月亮不可见）的日期定位月首，也就是“初一”，把月圆的时候定为望日，也就是“十五”，月亮绕地球公转一周称为一个朔望月。天文学的朔望月长度是 29.5306 日，中国农历以朔望月为基础，严格保证每个月的头一天是朔日，这就使得每个月是大月还是小月的安排不能固定，通常需要通过天文学观测和计算来确定。一个农历年由 12 个朔望月组成，这样一个农历年的长度就是 $29.5306 \times 12 = 354.3672$ 日，而阳历的一个天文学回归年是 365.2422 日，这样一个农历年就比一个回归年少 10.88 天，这个误差如果累计起来过 16 年就会出现“六月飞雪”的奇观了。为了协调农历年和回归年之间的矛盾，聪明的先人在天文观测的基础上，找到了“闰月”的方法，通过在适当的月份插入闰月来保证每个农历年的正月到三月是春季，四月到六月是夏季，七月到九月是秋季，十月到十二月是冬季，也就是说，让历法和天文气象能够基本对上，不至于出现“六月飞雪”。

那么多长时间增加一个闰月比较合适呢？最早人们推算是“三年一闰”，后来是“五年两闰”，随着历法计算的精确，最终定型为“十九年七闰”。这个“十九年七闰”又是怎么算出来的呢？其实就是求出回归年日数和朔望月日数的最小公倍数，也就是 m 个回归年的天数和 n 个朔望月的天数相等，即：

$$m \times 365.2422 = n \times 29.5306$$

这样 m 和 n 的比例就是 29.5306∶365.2422≈9∶235，按照这个最接近的整数倍数关系，每 19 个回归年需要添加的闰月就是：

$$235 - 12 \times 19 = 7$$

也就是“十九年七闰”的由来。但是需要注意的是，“十九年七闰”也并不是精确的结果，每 19 年就会有 0.0892 天的误差：

$$19 \times 365.2422 - 235 \times 29.5306 \approx 0.0892$$

这样每 213 年就会积累约 1 天的误差，因此，即使按照“十九年七闰”计算，中国农历每一两百年就需要修正一次。正因为这样，现行农历从唐代以后就已经不再遵守“十九年七闰”法，而是采用更准确的“中气置闰”法。“中气置闰”法更准确的名称应该是“定冬至”法，就是定两个冬至节气之间的时间为一个农历年，这样农历年的长度就和太阳回归年长度对应，不会产生误差。

现在，我们知道农历通过置闰月的方式协调农历年和回归年长度不相等的问题，也知道了置闰的方法是“中气置闰”法，那么到底什么是“中气”，又是如何定中气置闰月呢？要回答这个问题，就需要再来回顾一下 11.2.1 节介绍的一种天文现象——节气。由于节气在回归年中是均匀分布的，因此公历中的节气日期基本上是固定的，比如立春是在公历的 2 月 3 日到 5 日，不会超出这个日期范围，这也就是《二十四节气歌》所说的：每月两节不变更，最多相差一两天。但是在中国农历中哪个中气属于哪个月是有规定的，雨水是正月的中气，春分是二月的中气，谷雨是三月的中气，小满是四月的中气，夏至是五月的中气，大暑是六月的中气，处暑是七月的中气，秋分是八月的中气，霜降是九月的中气，小月是十月的中气，冬至是十一月的中气，大寒是十二月的中气。

传统上一个农历年起于冬至节气，结束于冬至节气，因此要确定在哪一年置闰，主要看那一年两个冬至之间有几个朔望月。如果两个冬至节气之间有 12 个朔望月，则不置闰，如果有十三个朔望月，则置闰月，至于闰几月，则要看节气而定。对于有 13 个朔望月的农历年，置闰月的规则就是从农历二月开始到十月，第一个没有中气的月就是闰月，这个没有中气的朔望月跟在哪个月后面就是闰几月。为什么会有没有中气的朔望月呢？黄道上两个中气之间相隔 30 度，一个回归年的长度是 365.2422 日，则两个中气之间的平均间隔是 $365.2422 \div 12 = 30.4368$ 日，但是因为地球轨道是椭圆轨道，因此相邻的两个中气的时间间隔是不均匀的，比如在远地点附近的中气间隔就会长一点，最长可能是 31.45 天。而农历的朔望月平均长度是 29.5306 日，这样就会出现

某个朔望月刚好落在两个中气之间的情况，比如，某个月的上一个月月末是一个中气，但是下一个中气落在这个月的下一个月的头几天里，这样这个月就没有中气了。举个例子，2001 年农历辛巳年的四月二十九（公历 5 月 21 日）是小满，农历四月之后的这个朔望月从公历 5 月 23 日持续到公历 6 月 20 日，而小满后的下一个中气夏至是在公历的 6 月 21 日，也就是农历四月的下下个月的初一，这样农历四月后的这个月就没有中气，跟在四月之后，就称为闰四月。

3. “月建”问题

在了解了农历与节气的关系以及农历如何置闰月的方法之后，还需要解决一个问题才能着手农历年历的推算，那就是如何确定农历年的开始，或者说哪个月的初一是农历新年的开始？要回答这个问题，就需要了解中国农历特有的“月建”问题。

中国农历是阴阳合历，需要同时考虑太阳和月亮的位置。所以在确定岁首（元旦）时，需要先确定它在某个季节，然后再选定与这个季节相近的朔望月作为岁首。由于一岁（一个回归年）和 12 个阴历月并不相等，相差约 10.88 天，因此每隔三年需要设置一个闰月调整季节。中国上古的天文学家想出了一个简便的方法判断月序与季节的关系，这就是以傍晚时北斗七星的斗柄的指向确定月序，称为“十二月建”。从北方起向东转，将地面划分为十二个方位，傍晚时北斗所指的方位，就是该月的月建，其子月为冬至所在之月，对应十一月，丑月是冬至所在之月的次月，对应十二月，寅月在丑月之后，对应正月。中国在历史上的不同时期，多次修改过岁首（元旦）的起始月份，上古时代就有“三正”之说，所谓“三正”，就是“夏正建寅、殷正建丑、周正建子”，意思是夏历以寅月（正月）为岁首，殷历以丑月（十二月）为岁首，周历以子月（十一月）为岁首。从秦代到西汉前期又采用秦历，秦历建亥，也就是以亥月作为岁首之月，汉武帝太初元年（公元前 104 年）改用太初历，重新适用建寅的夏历，以寅月（正月）为岁首。在这之后的两千多年时间里，除王莽和魏明帝一度改用建丑的殷历，唐武后和肃宗时改用建子的周历外，各个朝代均使用建寅的夏历直到清朝末年。辛亥革命胜利以后，南京国民政府将公历 1 月 1 日改为元旦，但是人们仍习惯称农历的正月初一为元旦。新中国成立初期召开的第一届政治协商会议，正式将公历的 1 月 1 日确定为元旦，将农历的正月初一定为“春节”，也就是说，农历的岁首仍然采用夏历从寅月（正月）开始。

4. 农历基本历法规则

了解了“月建”问题，就解决了农历朔望月与公历月的对应关系，那就是冬至节气所在的朔望月就是农历的子月，对于目前适用的夏历建寅的月建体系，就意味着冬至节气所在的朔望月是农历的十一月，只要找到这个朔望月的起始日（日月合朔发生的时刻所在的那一日），就找到了公历的日期月农历日期的对应关系。下面总结一下中国农历历法的基本法则。

(1) 严格以日月合朔发生时刻为月首，这一天定为初一，通过计算两次日月合朔的时间间隔确定每月是 29 天还是 30 天，29 天的月份为小月，30 天的月份为大月；

(2) 月以中气得名，冬至节气总是出现在农历十一月，包含雨水中气的月为正月（即寅月），月无中气者为闰月，与前一个月同名；

(3) 从某一年的冬至后第一天开始，到下一个冬至这段时间内，如果有十三个朔望月出现，则此期间要增加一个闰月，从二月到十月，第一个没有中气的月就是闰月，如果在此期间有超过两个朔望月没有中气，则只有第一个没有中气的朔望月是闰月；

(4) 农历年以正月初一为岁首（关于农历岁首的说法，可参见下一小节），以腊月（十二月）廿九或三十为除夕；

(5) 如果节气和日月合朔在同一天，则该节气是这个新朔望月的节气。（民间历法）

规则(5)对节气和朔日在同一天的处理，采用了民间历法的处理原则，关于民间历法和历理历法的区别，我们马上就会讲到。

5. 农历年和农历生肖年（正月初一和立春节气）

立春是二十四节气之首，所以古代民间都是在"立春"这一天过节，相当于现代的春节（中国古代即是节气也是节日的情况很多，比如清明、冬至等）。1911 年，孙中山领导的辛亥革命建立了中华民国，在从历法上正式把农历正月初一定为"春节"，把公历 1 月 1 日定为"元旦"，也就是"新年"。农历年从正月初一开始没有争议，但是农历生肖年从何时开始却一直有争议，目前多数人都认为"立春"节气是农历生肖年的开始。因为在中国古代历法中，十二生肖的计算与天干地支有很大关系，所以在"论天干地支、计算廿四节气"的情况下，"立春"节气应该是新生肖的开始。对于普通老百姓来说，习惯于认为正月初一是生肖年的开始，因此，正月初一和"立春"节气之间出生的小孩，在确定属相的时候就有点麻烦了。属马还是属羊？这是个问题。

6. 民间历法和历理历法

新中国成立以后没有颁布新的"官方农历历法"，将历法和政治分离体现了时代的进步，但是由于没有 "官方历法"，也引起了一些问题。比如我国现在采用的农历历法是《时宪历》，它源于清朝顺治年间（公元 1645）颁布的《顺治历》，它有两个不足之处：一个是日月合朔和节气的时间以北京当地时间为准，也就是东经 116 度 25 分的当地时间，其节气和新月的观察只适用于中原地区。其他经度的地方，因为时间的关系，对导致日月合朔和节气时间的差异导致置闰和月顺序各不相同。另一个不足之处就是日月合朔时间和节气时间判断不精确，如果日月合朔时间和节气时间在同一天，不管具体的时间是否有先后，一律将此节气算作新月中的节气，这样一来，如果这个节气是中气，就会影响到闰月的设置。历理历法针对这两点进行了改进，对节气时间和日月合朔时间统一采用东经 120 度即东八区标准时，这样在任何时区的节气和置闰结果都是一样的，以东八区标准时为准。对于节气时间和日月合朔时间在同一天的情况，精确计算到时、分、秒，只有日月合朔时间在节气时间之前，这个节气才包含在次月内。历理历法从理论上讲更符合现代天文学的精确计算，但是需要注意的是，历理历法仍然只是存在于理论上的历法，我国现行的农历历法依然是民间历法《时宪历》或《顺治历》。

11.4.2 中国农历的推算

了解了农历历法的基本法则后，就可以根据历法进行农历年历的推算。农历年历的推算是一

件很复杂的事情，需要知道每年二十四个节气和本年内每次日月合朔的精确时间，这些时间的获取比较困难。现在有很多可以显示农历的日历软件，其实并不计算这些时间，而是事先从权威机构（如紫金山天文台）获取这些经过推算的时间，然后用各种方法将这些信息存储在设计好的数据结构中。当计算农历时采用查表的方法获取每年的二十四节气日期、大小月情况以及闰月情况，这样的软件受数据量的限制，往往只能显示近一两百年的年历。

本章要介绍的方法是建立在之前介绍的天文算法的基础上的计算方法，不同于查表法，这种方法不需要任何事先预设的数据，可以计算任何年份的历法。当然，受很多条件的限制，这种方法也不是万能的。首先，历史上已经确定的事件，仍然要以历史为主。比如通过现代计算发现古人观测存在误差，比如某个月不是闰月，或者某个节气不是这一天，但仍然要按照历史已经记载的历法使用。其次，我们目前所使用的星历表和各种半解析理论，都只能在近两三千年的时间里将误差控制在一定范围内，更远的时间上，肯定需要新的理论进行修正或替换。因此，万年历是个伪命题，不存在一统万年的万年历，即使用天文计算的方法，也不能保证万年以后的准确性。地球自转的速度正在变慢，月亮正在以每年 1 厘米的速度远离地球，这些非周期性变化的因素都会导致万年以后的计算毫无意义。

忘掉万年历吧，本章的重点是介绍如何推算农历历法，即便使用的是先进理论指导下的天文算法，也不能保证任意时刻都是有意义的。

1. 利用经验值推算农历

在各种天文算法的理论出现之前，人们一般采用一些经验公式近似的计算农历。有一些经验公式可以用来计算节气发生的日期，也一些经验公式用来计算朔日。通式寿星公式是前人整理出来的一个用于计算每年立春日期的经验公式，可以计算出某一年的某个节气时间，但是只能精确到日。其定义如下：

$$\text{Date}=\lfloor Y\times D+C\rfloor - L$$

其中，Y 是年份，D 的值是 0.2422，C 是经验值，取决于节气和年份，对于 21 世纪，立春节气的 C 值是 4.475，春分节气的 C 值是 20.646。L 是闰年数，其计算公式为：

$$L=\lfloor Y/4\rfloor - \lfloor Y/100\rfloor + \lfloor Y/400\rfloor$$

用通式寿星公式确定 2011 年立春日期的过程如下：

$$L = \text{int}(2011/4) - \text{int}(2011/100) + \text{int}(2011/400) = 502-20 + 5 = 487$$

$$\text{Date} = \text{int}(2011\times0.2422+4.475) - 487 = 491 - 487 = 4$$

所以，2011 年的立春日期是 2 月 4 日。

历史上还有人给出了计算节气和朔日的积日公式，以 1900 年 1 月 0 日（星期日）为基准日，之后的每一天与基准日的差值称为*积日*，1900 年 1 月 1 日的积日是 1，以后的时间以此类推。则计算 1900 年之后第 y 年第 x 个节气的积日公式是：

$$F = 365.242 * (y - 1900) + 6.2 + 15.22 * x - 1.9 * \sin(0.262 * x)$$

其中 x 是节气的索引，0 代表小寒，1 代表大寒，其他节气按照顺序类推。计算从 1900 年开始第 m 个朔日的公式是：

$$M = 1.6 + 29.5306 * m + 0.4 * \sin(1 - 0.45058 * m)$$

以上两个公式计算的结果是从 1900 年 1 月 0 日开始的积日，需要根据这些年的闰年情况转化为具体年份的具体日期。毫无疑问，这两个公式也只能精确到日，并且随着时间距离 1900 年越远，误差越大，时至今日已经很少使用了。

还有一种确定节气时间和朔日时间的方法，就是在已知某个节气或朔日的精确时间后，通过某些规律先前或向后推算其他节气或朔日的时间。二十四个节气就是黄道上的 24 个点，由于地球运动受其他天体的影响，导致这些节气在每年的时间是不固定的，但是这些节气之间的间隔时间基本上可以看作是固定的，表 11-1 就是二十四节气的时间间隔表。

表11-1　二十四节气时间间隔表（单位：秒钟）

节气名	与上一节气之间的时间差	与小寒节气的累积时间差
小寒	1271448.00	0.00
大寒	1272494.40	1272494.40
立春	1275526.20	2548020.60
雨水	1282123.20	3830143.80
惊蛰	1290082.80	5120226.60
春分	1300639.20	6420865.80
清明	1311153.00	7732018.80
谷雨	1323253.80	9055272.60
立夏	1333685.40	10388958.00
小满	1344107.40	11733065.40
芒种	1351227.00	13084292.40
夏至	1357299.60	14441592.00
小暑	1358968.80	15800560.80
大暑	1358786.40	17159347.20
立秋	1354419.00	18513766.20
处暑	1348236.00	19862002.20
白露	1339003.20	21201005.40
秋分	1328654.40	22529659.80
寒露	1317185.40	23846845.20
霜降	1305760.80	25152606.00
立冬	1295081.40	26447687.40
小雪	1285764.00	27733451.40
大雪	1278469.80	29011921.20
冬至	1273556.40	30285477.60

已知 1900 年小寒时刻为 1 月 6 日 2:05:00，以这个节气时刻为基准，推算其他年份节气的算法实现如下：

```
static double s_stAccInfo[] =
{
    0.00, 1272494.40, 2548020.60, 3830143.80, 5120226.60, 6420865.80,7732018.80, 9055272.60,
10388958.00, 11733065.40, 13084292.40, 14441592.00,15800560.80, 17159347.20, 18513766.20, 19862002.20,
21201005.40, 22529659.80, 23846845.20, 25152606.00, 26447687.40, 27733451.40, 29011921.20, 30285477.60
};

//已知 1900 年小寒时刻为 1 月 6 日 02:05:00，
const double base1900_SlightColdJD = 2415025.5868055555;

double CalculateSolarTermsByExp(int year, int st)
{
    if((st < 0) || (st > 24))
        return 0.0;

    double stJd = 365.24219878 * (year - 1900) + s_stAccInfo[st] / 86400.0;

    return base1900_SlightColdJD + stJd;
}
```

base1900_SlightColdJD 是北京时间 1900 年 1 月 6 日凌晨 2:05:00 的儒略日数，CalculateSolarTermsByExp()函数返回指定年份的节气的儒略日数。已知某个朔日的精确时间推算其他朔日时间的方法也类似，以朔望月的长度为单位向前或向后累加即可。

这种推算的方法是建立在地球回归年的长度是固定 365.2422 天、节气的间隔是绝对固定的、朔望月长度是平均的 29.5305 天等假设之上的，由于天体运动的互相影响，这种假设不是绝对成立的，因此这种推算方法的误差很大。以 CalculateSolarTermsByExp()函数为例，计算 1900 年前后 30 年内的节气时间的误差还可以控制在 30 分钟以内，但是到 2000 年的时候误差已经超过 130 分钟了。

2. 根据天文算法精确推算农历

要想精确地获得几百年乃至更长时间范围内任意一年的节气发生时间和日月合朔时间，就只能采用“天文算法”。本章介绍了 VSOP82/87 太阳系行星运行理论和 ELP-2000/82 月球运行理论，这是两种半解析理论，也是本章所给出的算法的基础。如果要求更高的精度，可以考虑使用各天文台或研究机构发布的有针对性的星历表。比较著名的星历表有美国国家航空航天局下属的喷气推进实验室发布的 DE 系列星历表，还有瑞士天文台在 DE406 基础上拓展的瑞士星历表等。根据行星运行轨道直接计算行星位置通常不是很方便，更何况大多数民用天文计算用不上那么多精确的轨道参数，所以使用本章介绍的两个理论对于日历的推算已经够用了。

3. 农历与公历的对应关系

中国的官方纪时采用的是中国公历（格里历），因此农历年历的推导应以公历年的周期为主导，附上农历年的信息，也就是说，年历以公历的 1 月 1 日为起始，至 12 月 31 日结束，根据农历历法推导出的农历日期信息，附加在公历日期信息上形成双历。通常情况下，一个公历年周期都不能完整地对应到一个农历年周期上，二者的偏差也不固定，因此不存在稳定的对应关系，也

就是说，不存在从公历的日期到农历日期的转换公式，只能根据农历的历法规则推导出农历日期与公历日期的对应关系。由农历历法规则可知，上一个公历年的冬至所在的朔望月是上一个农历年的十一月（冬月），所以在进行节气计算时，需要计算包括上一年冬至节气在内的二十五个节气，才能对应上上一个农历年的十一月和当前农历年的十一月。在计算与之对应的朔日时，考虑到有闰月的情况，需要从上一年冬至节气前的第一个朔日，连续计算 15 个朔日才能保证覆盖两个冬至之间的一整年时间，图 11-4 显示了 2011 年没有闰月的情况下朔日和冬至的关系。

图 11-4 中上排数字是公历月的编号，黑色圆点代表朔日，黑色三角形代表冬至节气。图 11-5 显示了 2012 年有闰月的情况下朔日和冬至的关系。

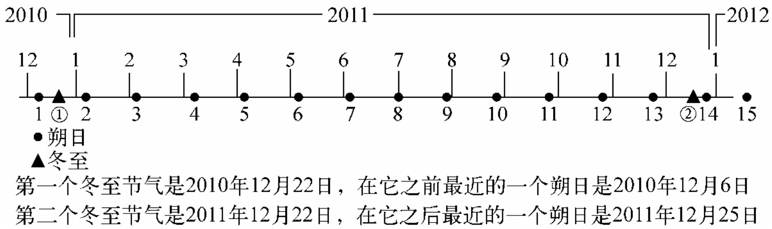


图 11-4　没有闰月情况下朔日与冬至节气关系图

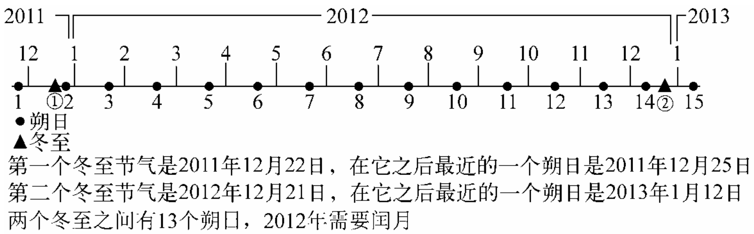


图 11-5　有闰月情况下朔日与冬至节气关系图

通过计算得到能够覆盖两个冬至节气的所有朔日时间后，就可以着手建立公历日期与农历日期的对应关系。以图 11-4 所示的 2011 年为例，首先根据计算得到的 15 个朔日（2011 年只会用到其中的前 14 个时间）时间，建立与 2011 年（公历年）有关的朔望月关系表（如表 11-2 所示）。

表11-2　2011年朔望月与公历日期关系表

朔日编号	合朔时间	对应公历日期	月　长	月　名
1	01:35:39.90	2010-12-06	29	冬月
2	17:02:34.26	2011-01-04	30	腊月
3	10:30:42.67	2011-02-03	30	正月
4	04:45:59.44	2011-03-05	29	二月
5	22:32:15.13	2011-04-03	30	三月
6	14:50:31.79	2011-05-03	30	四月
7	05:02:32.51	2011-06-02	29	五月

（续）

朔日编号	合朔时间	对应公历日期	月　　长	月　　名
8	16:53:54.10	2011-07-01	30	六月
9	02:39:45.06	2011-07-31	29	七月
10	11:04:06.43	2011-08-29	29	八月
11	19:08:50.09	2011-09-27	30	九月
12	03:55:54.64	2011-10-27	29	十月
13	14:09:40.97	2011-11-25	30	冬月
14	02:06:27.05	2011-12-25	29	腊月
15	15:39:23.99	2012-01-23	30	正月

两个朔望月之间有多少天，这个农历月就有多少天，29 天是小月，30 天是大月，分别冠以某月小或某月大的月名。在图 11-4 和图 11-5 中，编号为 1 和 2 的两个朔日之间的朔望月是冬月（十一月），因为冬至节气落在这个朔望月，其他月的月名以此类推，正月的朔日就是春节。输出公历和农历双历时，以月（公历）为单位，从每月第一天开始，依次判断每一天属于哪个朔望月，确定这一天的农历月名，然后比较这一天和这个朔望月的朔日之间相差几天，记为农历日期。以 2011 年 1 月 1 日为例，这一天在 2010 年 12 月 6 日（2010 年农历冬月的朔日）和 2011 年 1 月 4 日之间（2010 年农历腊月的朔日），查表 11-2 可知对应的农历月冬月，这一天和 2010 年 12 月 6 日相差 26 天，因此这一天的农历日期就是“廿七”。再以 2011 年 2 月 3 日（春节）这一天为例，查表 11-2 得知 2 月 3 日属于从 2 月 3 日开始的朔望月，这个朔望月的月名是正月，而 2 月 3 日就是月首，农历日期是初一，正月初一就是春节。

在 11.2 节和 11.3 节我们分别介绍了计算节气的 CalculateSolarTerms() 函数和计算日月合朔时间的 CalculateMoonShuoJD() 函数，现在就是用它们的时候了。生成指定公历年份的公历和农历的双历年历的算法流程如图 11-6 所示。

GetAllSolarTermsJD() 函数从指定年份的指定节气开始，连续计算 25 个节气时间，时间可以跨年份，内部判断过冬至节气后自动转到下一年的节气继续计算：

```
void CChineseCalendar::GetAllSolarTermsJD(int year, int start, double *SolarTerms)
{
    int i = 0;
    int st = start;
    while(i < 25)
    {
        SolarTerms[i++] = CalculateSolarTerms(year, st * 15);
        if(st == WINTER_SOLSTICE)
        {
            year++;
        }
        st = (st + 1) % SOLAR_TERMS_COUNT;
    }
}
```

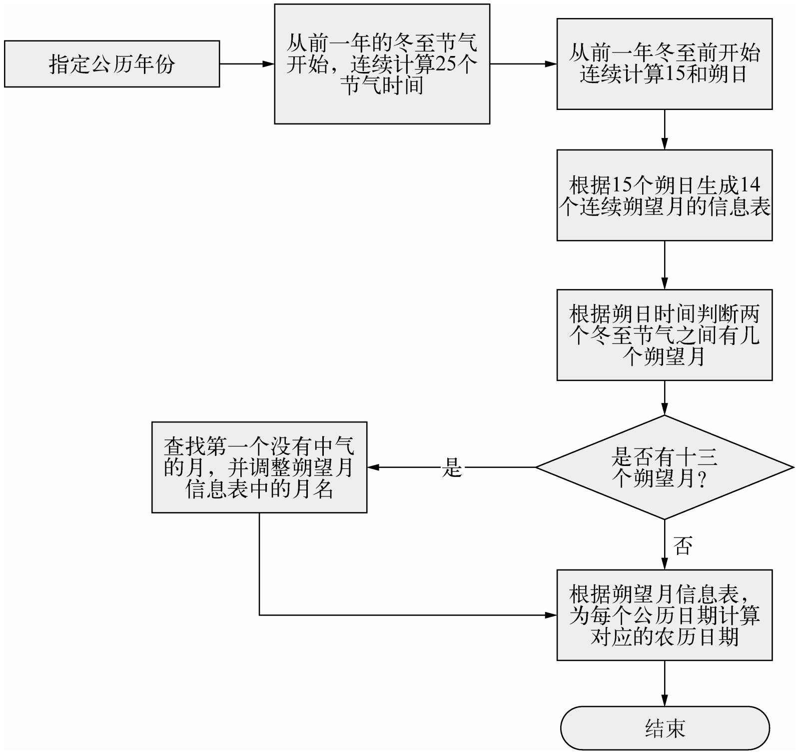


图 11-6 计算公农历双历年历的算法流程

其中 start 参数是节气的索引，定义二十四节气的索引如下：

```
const int VERNAL_EQUINOX      = 0;    // 春分
const int CLEAR_AND_BRIGHT    = 1;    // 清明
const int GRAIN_RAIN          = 2;    // 谷雨
const int SUMMER_BEGINS       = 3;    // 立夏
const int GRAIN_BUDS          = 4;    // 小满
const int GRAIN_IN_EAR        = 5;    // 芒种
const int SUMMER_SOLSTICE     = 6;    // 夏至
const int SLIGHT_HEAT         = 7;    // 小暑
const int GREAT_HEAT          = 8;    // 大暑
const int AUTUMN_BEGINS       = 9;    // 立秋
const int STOPPING_THE_HEAT   = 10;   // 处暑
const int WHITE_DEWS          = 11;   // 白露
const int AUTUMN_EQUINOX      = 12;   // 秋分
const int COLD_DEWS           = 13;   // 寒露
const int HOAR_FROST_FALLS    = 14;   // 霜降
const int WINTER_BEGINS       = 15;   // 立冬
const int LIGHT_SNOW          = 16;   // 小雪
const int HEAVY_SNOW          = 17;   // 大雪
const int WINTER_SOLSTICE     = 18;   // 冬至
const int SLIGHT_COLD         = 19;   // 小寒
const int GREAT_COLD          = 20;   // 大寒
```

11

```
const int SPRING_BEGINS      = 21;   // 立春
const int THE_RAINS          = 22;   // 雨水
const int INSECTS_AWAKEN     = 23;   // 惊蛰
```

节气索引乘以 15 就是节气在黄道上对应的度数。GetNewMoonJDs()函数从指定时间开始连续计算 15 个朔日时间，从第一个冬至节气前的第一个朔日开始。15 个朔日可以形成 14 个完整的朔望月，保证在有闰月的情况下也能包含两个冬至节气：

```
void CChineseCalendar::GetNewMoonJDs(double jd, double *NewMoon)
{
    double tdjd = JDLocalTimetoTD(jd);

    for(int i = 0; i < NEW_MOON_CALC_COUNT; i++)
    {
        NewMoon[i] = CalculateMoonShuoJD(tdjd);

        tdjd += 29.5; /*转到下一个最接近朔日的时间，牛顿迭代法的初始值*/
    }
}
```

BuildAllChnMonthInfo()函数根据 15 个朔日时间组成 14 个朔望月，根据相邻朔日的间隔计算出农历月天数用来判定大小月，并且从“十一月”开始依次为每个朔望月命名（月建名称）：

```
bool CChineseCalendar::BuildAllChnMonthInfo()
{
    CHN_MONTH_INFO info; //一年最多可个农历月
    int i;
    int yuejian = 11;   //采用夏历建寅，冬至所在月份为农历月
    for(i = 0; i < (NEW_MOON_CALC_COUNT - 1); i++)
    {
        info.mmonth = i;
        info.mname = (yuejian <= 12) ? yuejian : yuejian - 12;
        info.shuoJD = m_NewMoonJD[i];
        info.nextJD = m_NewMoonJD[i + 1];
        info.mdays = int(info.nextJD + 0.5) - int(info.shuoJD + 0.5);
        info.leap = 0;

        CChnMonthInfo cm(&info);
        m_ChnMonthInfo.push_back(cm);

        yuejian++;
    }

    return (m_ChnMonthInfo.size() == (NEW_MOON_CALC_COUNT - 1));
}
```

CalcLeapChnMonth()函数根据节气和朔日时间判断在两个冬至节气之间的农历年是否有闰月，判断的依据就是看第十四个朔日是否在第二个冬至节气之前，如果第十四个朔日发生在第二个冬至节气之前，就说明在两个冬至节气之间发生了十三次朔日，需要置闰月。因为农历中十二个中气属于哪个农历月是固定的，因此置闰月的过程就是依次判断十二个中气是否在对应的农历月中，如果本应该属于某个农历月的中气却没有落在这个农历月中，则这个农历月就是闰月，需要

设置闰月标志，同时调整这个月之后的月名。调整农历月名的方法就是月名减一，比如原来是八月就要调整为七月，这样就将十三个月对应上了十二个月名（其中多出来的一个农历月命名为闰某月）。如果节气和朔日发生在同一天，CalcLeapChnMonth()函数采用的是民间历法的规则，与现行历法一致：

```
/*根据节气计算是否有闰月，如果有闰月，根据农历月命名规则，调整月名称*/
void CChineseCalendar::CalcLeapChnMonth()
{
    assert(m_ChnMonthInfo.size() > 0); /*阴历月的初始化必须在这个之前*/

    int i;
    //第月的月末没有超过冬至，说明今年需要闰一个月
    if(int(m_NewMoonJD[13] + 0.5) <= int(m_SolarTermsJD[24] + 0.5))
    {
        //找到第一个没有中气的月
        i = 1;
        while(i < (NEW_MOON_CALC_COUNT - 1))
        {
            /*
             m_NewMoonJD[i + 1]是第 i 个农历月的下一个月的月首，本该属于第 i 个月的
             中气如果比下一个月的月首还晚，或者与下个月的月首是同一天（民间历法），
             则说明第 i 个月没有中气
            */
            if(int(m_NewMoonJD[i + 1] + 0.5) <= int(m_SolarTermsJD[2 * i] + 0.5))
                break;
            i++;
        }
        /*找到闰月，对后面的农历月调整月名*/
        if(i < (NEW_MOON_CALC_COUNT - 1))
        {
            m_ChnMonthInfo[i].SetLeapMonth(true);
            while(i < (NEW_MOON_CALC_COUNT - 1))
            {
                m_ChnMonthInfo[i++].ReIndexMonthName();
            }
        }
    }
}
```

11.4.3 一个简单的“年历”

到现在为止，我们已经计算出了一年之内所有的朔日和节气时间，也按照农历的规则准备好了农历月的信息（包括闰月）和与公历的对应关系，结合之前的天干地支和生肖信息，我们已经具备了一个日历所必需的全部元素，剩下的工作就是显示出这些信息，像日历一样展示给人们使用。我做了一个显示日历的小控件，将我们的成果展示出来，如图 11-7 所示，这就是我们本章的算法体现出的结果。



图 11-7　演示程序的界面

再遇到所谓的“万年历”软件的时候，你也不用感觉太神奇了，对于查表方式的软件，你可以藐视它们了。

11.5　总结

许多中国人有一种与生俱来的感觉，就是农历比公历准，实际上这种比较是没有意义的。年、月、日都是人类定义的记录单位，宇宙中的天体有自己的运行规律，才不管人类的感受。人类拿着自己定义好的年、月、日往上套，自然会出现误差，各种“闰”就是这么来的。农历采用“定冬至法”确定一年的区间，很好地解决了四季的气候变化与人类的主观感受之间的关系，与之对应的公历的月实际上就没有太大的意义，划分四季也有点牵强。但是不能根据这一点就说农历比公历准确。确切地说，农历和公历都不准，除非人类放弃年、月、日这种计时方式，不管昏天黑地和四季变化，统一采用一个标准计时单位记录时间。情侣们会这样微信他们的伴侣：“亲爱的，我在电影院门口等你，时间是第 10029384848737375 标准时间单位，不见不散！”至于第 10029384848737375 标准时间单位是冬天还是夏天，是白天还是晚上，估值也不会有人关注这个事情了，你愿意过这样的生活吗？

回到原来的话题，怎么记生日才准确，是农历，还是公历？都不是！你应该计算出你出生的那个时刻地球的日心黄经，记住这个位置，以后每当地球运行到这个位置时就庆祝你的生日吧。等等，我忘了什么事情了吗？对了，地球的自转，除了地球的日心黄经，还要计算地球自转偏转过的角度。流汗了吧？计算其实不难，难的是地球公转多少个周期才能刚好碰上这个偏角？你看，过个准确的生日这么难，这个生日你到底是过还是不过啊？

算法在生活中无处不在，有了各种行星运行理论，相关的天文计算的算法也简化了，非天文物理专业的历法爱好者也可以进行简单的历法计算，确定一些天文现象的时间。蔡勒公式让那些给个日期就能说出是星期几的神人不再神奇。当然，还有我们熟悉的牛顿迭代法，在计算二十四节气和朔日的时候两次用到了它。

11.6 参考资料

[1] Chapront-Touze M, Chapront J. *ELP 2000-85 - A Semi-Analytical Lunar Ephemeris Adequate for Historical Times*. Astronomy And Astrophysics, 1998

[2] Meeus J. *Astronomical Algorithms*. Willmann-Bell, Inc, 1991

[3] Secular Variations of the Planetary Orbits. http://www.worldlingo.com/ma/enwiki/en/Secular_ variations_of_the_planetary_orbits

[4] Bretagnon P, Francou G. *Planetray Theories in Rectangular and Spherical Variables VSOP87 Solutions*. Astronomy And Astrophysics, 1998

[5] Chapront-Touze M, Chapront J. *The Lunar Ephemeris ELP 2000*, Astronomy And Astrophysics, 1983

[6] Simon J L, Chapront-Touze M, Chapront J, et al. *Numerical Expression for Precession formulae and Mean Elements for the Moon and the Planets*. Astronomy And Astrophysics, 1994

[7] Chapront-Touze M, Chapront J. *Analytical Ephemerides of the Moon in the 20th Century*. Paris Observatory Lunar Analysis Center, 2002

[8] Chapront-Touze M. *New Expressions of the Secular Terms in Lunar Table and Programs from 4000 B.C. to A.D. 8000*. Paris Observatory Lunar Analysis Center,1999

第 12 章 实验数据与曲线拟合

在科学研究和工程实践过程中，都有大量的数据需要分析和处理。对这些数据进行图形化的展示，相对于一堆离散的数据来说，能更直观地看出数据的实际意义和变化趋势。将数据转换成图形展示有很多种方法，曲线拟合就是二维图形化展示的一种，本章就介绍两种最常见的曲线拟合算法。

12.1 曲线拟合

科学和工程上遇到的很多问题，往往只能通过诸如采样、实验等方法获得若干离散的数据，根据这些数据，如果能够找到一个连续的函数（也就是曲线）或者更加密集的离散方程，使得实验数据与方程的曲线能够在最大程度上近似吻合，就可以根据曲线方程对数据进行数学计算，对实验结果进行理论分析，甚至对某些不具备测量条件的位置的结果进行估算。

12.1.1 曲线拟合的定义

曲线拟合（curve fitting）的数学定义是指用连续曲线近似地刻画或比拟平面上一组离散点所表示的坐标之间的函数关系，是一种用解析表达式逼近离散数据的方法。曲线拟合通俗的说法就是“拉曲线”，也就是将现有数据透过数学方法来代入一条数学方程式的表示方法。由以上定义可知，曲线拟合不仅仅是根据离散数据画出一条曲线，曲线拟合更重要的意义是通过特定的曲线拟合算法，推算出一个（或一系列）能逼近离散数据并能维持统计误差最小的数学解析表达式，也就是拟合曲线的数学方程。

12.1.2 简单线性数据拟合的例子

回想一下中学物理课的“速度与加速度”实验：假设某物体正在做加速运动，加速度未知，某实验人员从时间 $t_0 = 3$ 秒时刻开始，以 1 秒时间间隔对这个物体连续进行了 12 次测速，得到一组速度和时间的离散数据（如表 12-1 所示），请根据实验结果推算该物体的加速度。

表12-1　物体速度和时间的测量关系表

时间（s）	3	4	5	6	7	8	9	10	11	12	13	14
速度（m/s）	8.41	9.94	11.58	13.02	14.33	15.92	17.54	19.22	20.49	22.01	23.53	24.47

在选择了合适的坐标刻度之后，我们就可以在坐标纸上画出这些点。如图 12-1 所示，排除偏差明显偏大的测量值后，可以看出测量结果呈现典型的线性特征。沿着该线性特征画一条直线，使尽量多的测量点位于直线上，或与直线的偏差尽量小，这条直线就是我们根据测量结果拟合的速度与时间的函数关系。最后在坐标纸上测量出直线的斜率 K，K 就是被测物体的加速度。经过测量，我们实验测到的物体加速度值是 1.53m/s^2，初速度是 3.99m/s。

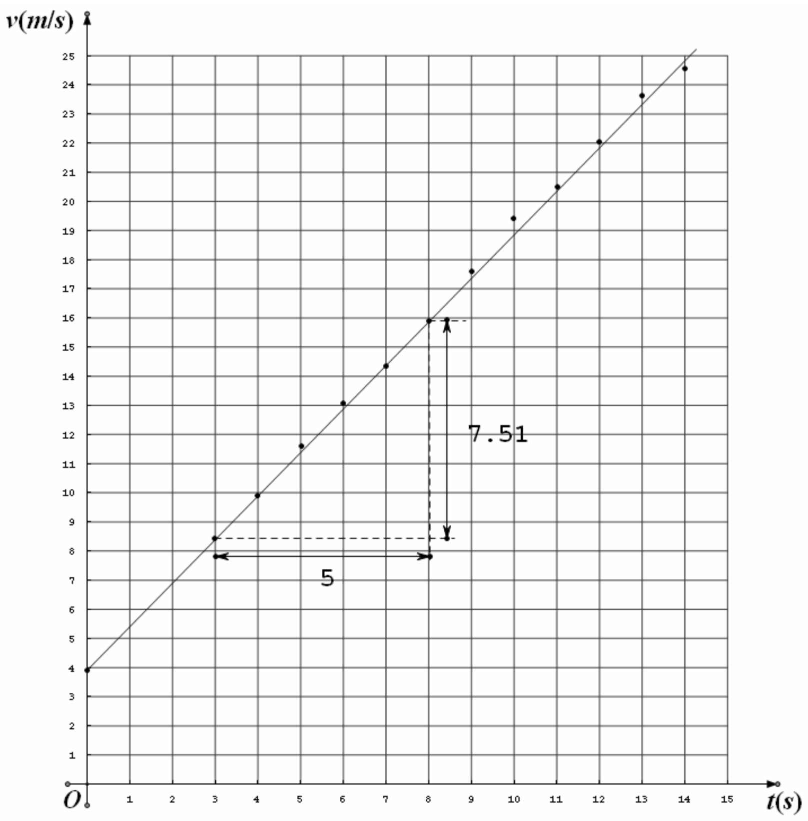


图 12-1　实验法测量加速度的过程

12.2　最小二乘法曲线拟合

使用数学分析进行曲线拟合有很多常用的方法，这一节我们先介绍一下最简单的最小二乘法，并给出使用最小二乘法解决上一节给出的速度与加速度实验问题。

12.2.1 最小二乘法原理

最小二乘法，又称最小平方法，是一种通过最小化误差的平方和寻找数据的最佳函数匹配的方法。利用最小二乘法，可以简便地求得未知的数据，并使得这些求得的数据与实际数据之间误差的平方和为最小。当然，作为一种插值方法使用时，最小二乘法也可以用于曲线拟合。使用最小二乘法进行曲线拟合是曲线拟合中早期的一种常用方法。不过，最小二乘法理论简单，计算量小。即便在使用三次样条曲线或 RBF（Radial Basis Function）进行曲线拟合大行其道的今天，最小二乘法在多项式曲线或直线的拟合问题上，仍然得到广泛的应用。使用最小二乘法，选取的匹配函数的模式非常重要：如果离散数据呈现的是指数变化规律，则应该选择指数形式的匹配函数模式；如果是多项式变化规律，则应该选择多项式匹配模式。如果选择的模式不对，拟合的效果就会很差，这也是使用最小二乘法进行曲线拟合时需要特别注意的一个地方。

下面以多项式模式为例，介绍一下使用最小二乘法进行曲线拟合的完整步骤。假设选择的拟合多项式模式是：

$$y = a_0 + a_1 x + a_2 x^2 + \cdots, + a_m x^m \tag{12-1}$$

离散的各点到这条曲线的平方和 $F(a_0,a_1, \cdots, a_m)$则为：

$$F(a_0, a_1, \cdots, a_m) = \sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m)]^2 \tag{12-2}$$

最小二乘法的第一步处理就是对 $F(a_0,a_1,\cdots, a_m)$ 分别求对 a_i 的偏导数，得到 m 个等式：

$$-2\sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m)] = 0$$

$$-2\sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m)]x_i = 0$$

$$\cdots$$

$$-2\sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m)]x_i^m = 0 \tag{12-3}$$

这 m 个等式相当于 m 个方程，$a_0,a_1,\cdots,a_m$ 是 m 个未知量，因此这 m 个方程组成的方程组是可解的，最小二乘法的第二步处理就是将其整理为针对 $a_0,a_1,\cdots,a_m$ 的正规方程组。最终整理的方程组如下：

$$a_0 n + a_1 \sum_{i=1}^{n} x_i + a_2 \sum_{i=1}^{n} x_i^2 + \cdots + a_m \sum_{i=1}^{n} x_i^m = \sum_{i=1}^{n} y_i$$

$$a_0 \sum_{i=1}^{n} x_i + a_1 \sum_{i=1}^{n} x_i^2 + a_2 \sum_{i=1}^{n} x_i^3 + \cdots + a_m \sum_{i=1}^{n} x_i^{m+1} = \sum_{i=1}^{n} x_i y_i$$

…

$$a_0\sum_{i=1}^{n}x_i^{\ m}+a_1\sum_{i=1}^{n}x_i^{\ m+1}+a_2\sum_{i=1}^{n}x_i^{\ m+2}+\cdots+a_m\sum_{i=1}^{n}x_i^{\ 2m}=\sum_{i=1}^{n}x_i^{\ m}y_i \tag{12-4}$$

最小二乘法的第三步处理就是求解这个多元一次方程组，得到多项式的系数 a_0，a_1，…，a_m，就可以得到曲线的拟合多项式函数。求解多元一次方程组的方法很多，高斯消元法是最常用的一种方法，下一节就简单介绍一下这种方法。

12.2.2 高斯消元法求解方程组

在数学上，高斯消元法是线性代数中的一个算法，可用来求解多元一次线性方程组，也可以用来求矩阵的秩，以及求可逆方阵的逆矩阵。高斯消元法虽然以数学家高斯的名字命名，但是最早出现在文献资料中应该是中国的《九章算术》。

高斯消元法的主要思想是通过对系数矩阵进行行变换，将方程组的系数矩阵由对称矩阵变为三角矩阵，从而达到消元的目的，最后通过回代逐个获得方程组的解。在消元的过程中，如果某一行的对角线元素的值太小，在计算过程中就会出现很大的数除以很小的数的情况，有除法溢出的可能，因此在消元的过程中，通常都会增加一个主元选择的步骤，通过行交换操作，将当前列绝对值最大的行交换到当前行位置，避免了除法溢出的问题，增加了算法的稳定性。

高斯消元法的实现简单，主要由两个步骤组成：第一个步骤就是通过选择主元，逐行消元，最终形成方程组系数矩阵的三角矩阵形式；第二个步骤就是逐步回代的过程，最终矩阵的对角线上的元素就是方程组的解。下面就给出高斯消元法的一个算法实现：

```
/*带列主元的高斯消去法解方程组，最后的解在 matrixA 的对角线上*/
bool GuassEquation::Resolve(std::vector<double>& xValue)
{
    assert(xValue.size() == m_DIM);

    /*消元，得到上三角阵*/
    for(int i = 0; i < m_DIM - 1; i++)
    {
        /*按列选主元*/
        int pivotRow = SelectPivotalElement(i);
        if(pivotRow != i)/*如果有必要，交换行*/
        {
            SwapRow(i, pivotRow);
        }
        if(IsPrecisionZero(m_matrixA[i * m_DIM + i]))/*主元是 0，不存在唯一解*/
        {
            return false;
        }
        /*对系数归一化处理，使每行的第一个系数是 1.0*/
        SimplePivotalRow(i, i);
        /*逐行进行消元*/
        for(int j = i + 1; j < m_DIM; j++)
        {
```

```
            RowElimination(i, j, i);
        }
    }
    /*回代求解*/
    m_matrixA[(m_DIM - 1) * m_DIM + m_DIM - 1] = m_bVal[m_DIM - 1] / m_matrixA[(m_DIM - 1) * m_DIM +
        m_DIM - 1];
    for(int i = m_DIM - 2; i >= 0; i--)
    {
        double totalCof = 0.0;
        for(int j = i + 1; j < m_DIM; j++)
        {
            totalCof += m_matrixA[i * m_DIM + j] * m_matrixA[j * m_DIM + j];
        }
        m_matrixA[i * m_DIM + i] = (m_bVal[i] - totalCof) / m_matrixA[i * m_DIM + i];
    }

    /*将对角线元素的解逐个存入解向量*/
    for(int i = 0; i < m_DIM; i++)
    {
        xValue[i] = m_matrixA[i * m_DIM + i];
    }

    return true;
}
```

在 GuassEquation::Resolve()函数中，m_matrixA 是以一维数组形式存放的系数矩阵，m_DIM 是矩阵的维数，SelectPivotalElement()函数从系数矩阵的第 i 列中选择绝对值最大的那个值所在的行，并返回行号，SwapRow()函数负责交换系数矩阵两个行的所有值，SimplePivotalRow()函数是归一化处理函数，通过除法操作将指定的行的对角线元素变换为 1.0，以便简化随后的消元操作。

12.2.3 最小二乘法解决“速度与加速度”实验

根据 12.2.1 节对最小二乘法原理的分析，用程序实现最小二乘法曲线拟合的算法主要由两个步骤组成，第一个步骤就是根据给出的测量值生成关于拟合多项式系数的方程组，第二个步骤就是解这个方程组，求出拟合多项式的各个系数。根据对上文最终整理的正规方程组的分析，可以看出其系数有一定的关系，就是每一个方程式都比前一个方程式多乘了一个 x_i。因此，只需要完整计算出第一个方程式的系数，其他方程式的系数只是将前一个方程式的系数依次左移一位，然后单独计算出最后一个系数就可以了，此方法可以减少很多无谓的计算。求解多元一次方程组的方法就使用 12.2.2 节介绍的高斯消元法，其算法上一节已经给出。

这里给出一个最小二乘算法的完整实现，以 12.1.2 节的数据为例，因为数据结果明显呈现线性方程的特征，因此选择拟合多项式为 $v = v_0 + at$，v_0 和 a 就是要求解的拟合多项式系数：

```
Bool LeastSquare(const std::vector<double>& x_value, const std::vector<double>& y_value, int M,
std::vector<double>& a_value)
{
    assert(x_value.size() == y_value.size());
    assert(a_value.size() == M);
```

```
    double *matrix = new double[M * M];
    double *b= new double[M];

    std::vector<double> x_m(x_value.size(), 1.0);
    std::vector<double> y_i(y_value.size(), 0.0);
    for(int I = 0; I < M; i++)
    {
        matrix[ARR_INDEX(0, I, M)] = std::accumulate(x_m.begin(), x_m.end(), 0.0);
        for(int j = 0; j < static_cast<int>(y_value.size()); j++)
        {
            y_i[j] = x_m[j] * y_value[j];
        }
        b[i] = std::accumulate(y_i.begin(), y_i.end(), 0.0);
        for(int k = 0; k < static_cast<int>(x_m.size()); k++)
        {
            x_m[k] *= x_value[k];
        }
    }
    for(int row = 1; row < M; row++)
    {
        for(int I = 0; I < M - 1; i++)
        {
            matrix[ARR_INDEX(row, I, M)] = matrix[ARR_INDEX(row - 1, I + 1, M)];
        }
        matrix[ARR_INDEX(row, M - 1, M)] = std::accumulate(x_m.begin(), x_m.end(), 0.0);
        for(int k = 0; k < static_cast<int>(x_m.size()); k++)
        {
            x_m[k] *= x_value[k];
        }
    }

    GuassEquation equation(M, matrix, b);
    delete[] matrix;
    delete[] b;

    return equation.Resolve(a_value);
}
```

将表 12-1 的数据带入算法，计算得到 $v_0 = 4.05545455$，$a = 1.48818182$，比作图法得到的结果更精确。以上算法是根据最小二乘法的理论推导系数方程，并求解系数方程得到拟合多项式的系数的一种实现方法。除此之外，还可以利用预先计算好的最小二乘解析理论直接求得拟合多项式的系数，读者可自行学习相关的实现算法。

12.3 三次样条曲线拟合

曲线拟合基本上就是一个插值计算的过程，除了最小二乘法，其他插值方法也可以被用于曲线拟合。常用的曲线拟合方法还有基于 RBF 的曲线拟合和三次样条曲线拟合。最小二乘法方法简单，便于实现，但是如果拟合模式选择不当，会产生较大的偏差，特别是对于复杂曲线的拟合，如果选错了模式，拟合的效果就很差。基于 RBF 的曲线拟合方法需要高深的数学基础，涉及多

维空间理论，将低维的模式输入数据转换到高维空间中，使得低维空间内的线性不可分问题在高维空间内变得线性可分，这种数学分析方法非常强大，但是这种方法不宜得到拟合函数，因此在需要求解拟合函数的情况下使用起来不是很方便。

样条插值是一种工业设计中常用的、得到平滑曲线的一种插值方法，三次样条又是其中用得较为广泛的一种。使用三次样条曲线进行曲线拟合可以得到非常高精度的拟合结果，并且很容易得到拟合函数，本节的内容将重点介绍三次样条曲线拟合的原理和算法实现，并通过一个具体的例子将三次样条函数拟合的曲线与原始曲线对比显示，让大家体会一下三次样条曲线拟合的惊人效果。

12.3.1 插值函数

前面提到过，曲线拟合的实质就是各种插值计算，因此，插值函数的选择决定了曲线拟合的效果。那么插值函数的数学定义是什么呢？若在$[a, b]$上给出 $n+1$ 个点 $a \leqslant x_0 < x_1 < \cdots < x_n \leqslant b$，$f(x)$ 是$[a, b]$上的实值函数，要求一个具有 $n+1$ 个参量的函数 $s(x; a_0, \cdots, a_n)$ 使它满足

$$s(x_i; a_0, \cdots, a_n) = f(x_i)\ , i = 0, 1, \cdots, n \tag{12-5}$$

则称 $s(x)$为 $f(x)$ 在$[a, b]$上的插值函数. 若 $s(x)$ 关于参量 $a_0, a_1, \cdots, a_n$是线性关系，即：

$$s(x) = a_0 s_0(x) + a_1 s_1(x) + \cdots + a_n s_n(x) \tag{12-6}$$

$s(x)$就是多项式插值函数，如果 $s_i(x)$是三角函数，则 $s(x)$就是三角插值函数。

比较常用的多项式插值函数是牛顿插值多项式和拉格朗日插值多项式，但是在多项式的次数比较高的情况下，插值点数 n 过多会导致多项式插值在收敛性和稳定性上失去保证，因此，当插值点数 n 较大的情况下，一般不使用多项式插值，而采用样条插值或次数较低的最小二乘法插值。

12.3.2 样条函数的定义

在所有能够保证收敛性和稳定性的插值函数中，最常用也是最重要的插值函数就是样条插值函数。采用样条函数计算出的插值曲线和曲面在飞机、轮船和汽车等精密机械设计中都得到了广泛的应用。样条插值函数的数学定义如下。

设区间$[a, b]$上选取 $n-1$ 个节点（包括区间端点 a 和 b 共 $n+1$ 个节点），将其划分为 n 个子区间 $a = x_0 < x_1 < \cdots < x_n = b$，如果存在函数 $s(x)$，使得 $s(x)$满足以下两个条件：

(1) $s(x)$ 在整个区间$[a, b]$上具有 $m-1$ 阶连续导数；

(2) $s(x)$在每个子区间$[x_{i-1}, x_i]$, $i = 1, 2, \cdots, n$ 上是 m 次代数多项式（最高次数为 m 次）；

则称 $s(x)$是区间$[a, b]$上的 m 次样条函数。假如区间$[a, b]$上存在实值函数 $f(x)$，使得每个节点处的值 $f(x_i)$与 $s(x_i)$相等，即

$$s(x_i) = f(x_i),\ i = 0, 1, \cdots, n \tag{12-7}$$

则称 $s(x)$是实值函数 $f(x)$的 m 次样条插值函数。

当 $m = 1$ 时，样条插值函数就是分段线性插值，此时虽然 $s(x)$是属于区间$[a, b]$上的函数，但它不光滑（连一阶连续导数性质都不具备），不能满足工程设计要求。工程设计通常使用较多的是 $m = 3$ 时的三次样条插值函数，此时样条函数具有二阶连续导数性质。

根据三次样条函数的定义，$s(x)$在每个子区间上的样条函数 $s_i(x)$都是一个三次多项式，也就是说，三次样条函数 $s(x)$由 n 个区间上的 n 个三次多项式组成，每个三次多项式可描述为以下形式：

$$s_i(x) = a_ix^3 + b_ix^2 + c_ix + d_i \qquad i = 1, 2,\cdots, n \tag{12-8}$$

因此，要确定完整的样条函数 $s(x)$需要确定 a_i、b_i、c_i和 d_i公 $4n$ 个系数。根据样条函数的定义，$s(x)$在区间内的 $n - 1$ 个节点处都是连续的，并且其一阶导数 $s_i'(x)$和二阶导数 $s_i''(x)$都是连续的，根据连续函数的性质（x_i的左右导数相等），我们可以得到 $3(n - 1)$个条件：

$$s_i(x_i - 0) = s_{i+1}(x_i + 0) \quad i = 1, 2, \cdots, n-1$$

$$s_i'(x_i - 0) = s_{i+1}'(x_i + 0) \quad i = 1, 2, \cdots, n-1$$

$$s_i''(x_i - 0) = s_{i+1}''(x_i + 0) \quad i = 1, 2, \cdots, n-1 \tag{12-9}$$

再加上插值函数在包括区间端点 a（就是 x_0），b（就是 x_n）在内的 $n + 1$ 个节点处满足 $s(x_i) = f(x_i)$，又可以得到 $n + 1$ 个条件，这样就具备了 $4n - 2$ 个条件。

12.3.3 边界条件

为了解决 $4n$ 个系数组成的方程组，最终确定的 $s(x)$，需要再补充两个边界条件使之满足 $4n$ 个条件。常用的边界条件有以下几种。

第一类边界条件，即满足 $s'(x_0) = f'(x_0)$，$s'(x_n) = f'(x_n)$两个条件，其中 $f(x)$是实值函数。

第二类边界条件，即满足 $s''(x_0) = f''(x_0)$，$s''(x_n) = f''(x_n)$两个条件，其中 $f(x)$是实值函数。特别情况下，当 $f''(x_0) = f''(x_n) = 0$ 的时候，也就是 $s''(x_0) = s''(x_n) = 0$ 的时候，第二类边界条件又被称为自然边界条件。

当样条函数的实值函数 $f(x)$是以$[a, b]$为周期的周期函数时，三次样条函数 $s(x)$在两个端点处满足 $s'(x_0 - 0) = s'(x_n + 0)$和 $s''(x_0 - 0) = s''(x_n + 0)$，这种情况又称为第三类边界条件。

工程技术中常用的是第一类边界条件和第二类边界条件，以及第二类边界条件的特殊情况自然边界条件。理想情况下，也就是实值函数已知的情况下，可以通过实值函数直接计算出边界条件的值，否则的话，就只能通过测量和计算得到边界条件的值，有时候甚至只能给出经验估计值，工程技术中通常根据实际情况灵活使用各类边界条件。

12.3.4 推导三次样条函数

求三次样条插值函数 $s(x)$的方法很多，其基本原理都是首先求出由待定系数组成的 $s(x)$，以及其一阶导数 $s'(x)$和二阶导数 $s''(x)$，然后将其带入到 12.3.2 节和 12.3.3 节列举的 $4n$ 个条件中，得到关于待定系数的方程组，最后求解方程组得到待定系数，并最终确定插值函数 $s(x)$。

求三次样条插值函数 $s(x)$常用的方法是“三转角法”和“三弯矩法”。根据三次样条函数的性质，$s(x)$的一阶导数 $s'(x)$是二次多项式， 二阶导数 $s''(x)$是一次多项式（线性函数），“三转角法”和“三弯矩法”的主要区别是利用这两个特性推导插值函数 $s(x)$、$s'(x)$和 $s''(x)$的方式不同。“三转角法”利用 $s(x)$的一阶导数 $s'(x)$是二次多项式这个特性，对于子区间$[x_i, x_{i+1}]$，利用抛物线插值公式获得一个通过 x_i 和 x_{i+1} 两个点的二次多项式作为 $s'(x)$，然后对 $s'(x)$进行积分和微分（求导）运算，分别得到 $s(x)$，和 $s''(x)$，最后将它们带入 $4n$ 个条件中求解系数方程组。“三弯矩法”则是利用 $s(x)$的二阶导数 $s''(x)$是一次多项式（线性函数）这个特性，对于子区间$[x_i, x_{i+1}]$，首先假设一个通过 x_i 和 x_{i+1} 两个点的线性函数作为 $s''(x)$，然后对 $s''(x)$进行连续两次积分运算得到 $s(x)$，再对 $s(x)$进行求导得运算到 $s'(x)$，最后将它们带入 $4n$ 个条件中求解系数方程组。这两种方法的本质是一样的，只是对 $s(x)$的推导过程不同，接下来就介绍使用“三弯矩法”求解三次样条函数的方法。

三次样条函数的求解过程就是系数方程组的推导过程，使用“三弯矩法”推导系数方程组，首先要确定插值函数的二阶导数 $s''(x)$。根据三次样条函数的性质，在每个子区间$[x_i, x_{i+1}]$上，其二阶导数 $s''(x)$是个线性方程，现在假设在 x_i 和 x_{i+1} 两个端点的二阶导数值分别是 M_i 和 M_{i+1}，也就是 $s''(x_i) = M_i$，$s''(x_{i+1}) = M_{i+1}$，则经过 x_i 和 x_{i+1} 的两点式直线方程是：

$$\frac{y-M_i}{x-x_i}=\frac{M_{i+1}-M_i}{x_{i+1}-x_i} \tag{12-10}$$

经过变换可以得到 $s_i''(x)$

$$y=s_i''(x)=\frac{x_{i+1}-x}{h_i}M_i+\frac{x-x_i}{h_i}M_{i+1} \qquad \text{其中 } h_i = x_{i+1} - x_i \tag{12-11}$$

对 $s_i''(x)$进行两次积分，得到 $s_i(x)$，其中 A_i 和 B_i 都是常量：

$$s_i(x)=\frac{(x_{i+1}-x)^3}{6h_i}M_i+\frac{(x-x_i)^3}{6h_i}M_{i+1}+A_i x+B_i \quad \text{其中 } h_i = x_{i+1} - x_i \tag{12-12}$$

根据式(12-7)插值条件，$s_i(x_i) = y_i$，$s_i(x_{i+1}) = y_{i+1}$，将这两个条件带入到式(12-12)，得到两个等式，这两个等式恰好是一个关于 A_i 和 B_i 的二元一次方程组：

$$\frac{(x_{i+1}-x_i)^3}{6h_i}M_i+A_i x_i+B_i=y_i$$

$$\frac{(x_{i+1}-x_i)^3}{6h_i}M_{i+1}+A_i x_{i+1}+B_i=y_{i+1} \tag{12-13}$$

因为 $h_i = x_{i+1} - x_i$，带入式(12-13)后简化等式，并求解这个方程组，得到 A_i 和 B_i 分别是：

$$A_i = \frac{y_{i+1} - y_i}{h_i} - \frac{M_{i+1} - M_i}{6} h_i$$

$$B_i = y_{i+1} - \frac{M_{i+1}}{6} h_i^2 - \left(\frac{y_{i+1} - y_i}{h_i} - \frac{M_{i+1} - M_i}{6} h_i\right) x_{i+1} \tag{12-14}$$

将 A_i 和 B_i 带入式(12-12)，得到完整的 $s_i(x)$：

$$s_i(x) = \frac{(x_{i+1} - x)^3}{6h_i} M_i + \frac{(x - x_i)^3}{6h_i} M_{i+1} + \left(y_i - \frac{M_i}{6} h_i^2\right) \frac{x_{i+1} - x}{h_i} + \left(y_{i+1} - \frac{M_{i+1}}{6} h_i^2\right) \frac{x - x_i}{h_i} \tag{12-15}$$

其中只有 M_i 和 M_{i+1} 是未知的系数量，只要求得 M_i 和 M_{i+1} 的值，就能够确定完整的样条函数 $s(x)$。要求解 M_i 和 M_{i+1}，还需要利用三次样条函数的一阶导函数的一些性质增加一些计算条件，因此还要求其一阶导函数 $s_i'(x)$。只需对 $s_i(x)$ 求导，就可以得到 $s_i(x)$ 的一阶导数 $s_i'(x)$：

$$s_i'(x) = -\frac{(x_{i+1} - x)^2}{2h_i} M_i + \frac{(x - x_i)^2}{2h_i} M_{i+1} + \frac{y_{i+1} - y_i}{h_i} - \frac{M_{i+1} - M_i}{6} h_i \tag{12-16}$$

根据三次样条函数的特性，其一阶导数 $s_i'(x)$ 在节点 x_i 处是连续的，就可以利用式(12-9)的第二个条件，即 $s_i'(x)$ 在节点 x_i 处左右导数相等的特性，再获得一些求解关于 M_i 的条件。根据左导数的定义：

$$s_i'(x_i - 0) = \frac{h_{i-1}}{6} M_{i-1} + \frac{y_i - y_{i-1}}{h_{i-1}} + \frac{h_{i-1}}{3} M_i \quad i = 1,2,\cdots,n-1 \tag{12-17}$$

同样，根据右导数的定义：

$$s_{i+1}'(x_i + 0) = -\frac{h_i}{6} M_i + \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{6} M_{i+1} \quad i = 1,2,\cdots,n-1 \tag{12-18}$$

由式(12-17)和式(12-18)可以得到一个等式，将 M_{i-1}、M_i 和 M_{i+1} 做为变量，将等式整理成关于 M_i 的方程：

$$\frac{h_{i-1}}{h_{i-1} + h_i} M_{i-1} + 2M_i + \frac{h_i}{h_{i-1} + h_i} M_{i+1} = \frac{6}{h_{i-1} + h_i}\left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}\right) \qquad i = 1,2,\cdots,n-1 \tag{12-19}$$

令 $u_i = \dfrac{h_{i-1}}{h_{i-1} + h_i}$，$v_i = 1 - u_i = \dfrac{h_i}{h_{i-1} + h_i}$，$d_j = \dfrac{6}{h_{i-1} + h_i}\left(\dfrac{y_{i+1} - y_i}{h_i} - \dfrac{y_i - y_{i-1}}{h_{i-1}}\right)$，将其带入式(12-19)，得到简化的等式：

$$u_i M_{i-1} + 2M_i + v_i M_{i+1} = d_i \quad i = 1,2,\cdots,n-1 \tag{12-20}$$

从 M_0 到 M_n 有 $n+1$ 个 M_i 的值需要求解，但是式(12-20)只有 $n-1$ 个等式，此时就需要用到两

个边界条件了。

如果是使用第二类边界条件，则直接可以得到以下两个条件等式：

$$s''(x_0) = M_0 = f''(x_0) = y_0' \tag{12-21}$$

$$s''(x_n) = M_n = f''(x_n) = y_n' \tag{12-22}$$

令 $d_0 = 2y_0'$，$d_n = 2y_n'$，可以得到由第二类边界条件确定的两个方程：

$$2M_0 = d_0 \tag{12-23}$$

$$2M_n = d_n \tag{12-24}$$

如果是使用第一类边界条件，即 $s'(x_0) = f'(x_0)$，$s'(x_n) = f'(x_n)$两个条件，则需要将这两个条件代入式(12-16)，通过计算得到两个条件等式。将 $s'(x_0) = y_0'$代入式(12-16)，得到：

$$2M_0 + M_1 = \frac{6}{h_0}\left(\frac{y_1 - y_0}{h_0} - y_0'\right) \tag{12-25}$$

将 $s'(x_n) = y_n'$代入式(12-16)，得到：

$$M_{n-1} + 2M_n = \frac{6}{h_{n-1}}\left(y_n' - \frac{y_n - y_{n-1}}{h_{n-1}}\right) \tag{12-26}$$

令 $d_0 = \frac{6}{h_0}\left(\frac{y_1 - y_0}{h_0} - y_0'\right)$，$d_n = \frac{6}{h_{n-1}}\left(y_n' - \frac{y_n - y_{n-1}}{h_{n-1}}\right)$，可将式(12-23)和式(12-24)简化为：

$$2M_0 + M_1 = d_0 \tag{12-27}$$

$$M_{n-1} + 2M_n = d_n \tag{12-28}$$

将第二类边界条件得到的式(12-23)和式(12-24)或第一类边界条件得到的式(12-27)和式(12-28)与式(12-20)中的 n–1 个等式组合在一起就得到一个关于 M_i 的方程组，求解此方程组可以得到 M_i 的值,代入到式(12-15)即可得到三次样条函数方程。以第一类边界条件得到的式(12-27)和式(12-28)为例，与式(12-20)连立得到以下方程组：

$$\begin{bmatrix} 2 & v_0 & & & \\ u_1 & 2 & v_1 & & \\ & \vdots & \vdots & \vdots & \\ & & u_{n-1} & 2 & v_{n-1} \\ & & & u_n & 2 \end{bmatrix} \begin{bmatrix} M_0 \\ M_1 \\ \vdots \\ M_{n-1} \\ M_n \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \tag{12-29}$$

这就是三弯矩方程组，其中 M_i，i=0,1,⋯, n 就是三次样条函数 $s(x)$的矩。根据式(12-27)和式(12-28)，$u_n = 1$，$v_0 = 1$，其余各系数可以通过式(12-19)中的系数计算出来。这个方程组的系数矩阵是一个对角线矩阵，并且是一个严格对角占优的对角阵（u_i 和 v_i 的值均小于主对角线的值，也就是 u_i 和 v_i 的值皆小于 2），可以使用追赶法求解。下一节将介绍如何使用追赶法求解方程组，

并给出求解的算法实现。

12.3.5 追赶法求解方程组

任意矩阵 A 都可以通过克洛脱（Crout）分解得到两个三角矩阵：

$$A=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}=\begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & \vdots & \vdots & \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix}\begin{bmatrix} 1 & u_{12} & \cdots & u_{1n} \\ & 1 & \cdots & u_{2n} \\ & & \vdots & \vdots \\ & & & 1 \end{bmatrix}=LU$$

，如果 A 是对角矩阵，则克洛脱分解的结果为：

$$A=\begin{bmatrix} a_1 & c_1 & & & & \\ b_2 & a_2 & c_2 & & & \\ & b_3 & a_3 & c_3 & & \\ & & \vdots & \vdots & \vdots & \\ & & & b_{n-1} & a_{n-1} & c_{n-1} \\ & & & & b_n & a_n \end{bmatrix}=\begin{bmatrix} l_1 & & & & & \\ m_2 & l_2 & & & & \\ & m_3 & l_3 & & & \\ & & \vdots & \vdots & & \\ & & & m_{n-1} & l_{n-1} & \\ & & & & m_n & l_n \end{bmatrix}\begin{bmatrix} 1 & u_1 & & & & \\ & 1 & u_2 & & & \\ & & 1 & u_3 & & \\ & & & \vdots & \vdots & \\ & & & & 1 & u_{n-1} \\ & & & & & 1 \end{bmatrix}$$

在分解后的矩阵中，$l_1=a_1$，$u_1=c_1/l_1$，其余各项的计算规则如下：

$$\begin{cases} m_i=b_i, i=2,3,\cdots,n \\ l_i=a_i-m_iu_{i-1}, i=2,3,\cdots,n \\ u_i=c_i/l_i, i=2,3,\cdots,n \end{cases}$$

在得到了各个系数后，原方程组就可以分解为两个方程组，即：$Ax=d\Rightarrow\begin{cases} Ly=d \\ Ux=y \end{cases}$，对于第一个方程，求解向量 y_i：

$$\begin{bmatrix} l_1 & & & & & \\ m_2 & l_2 & & & & \\ & m_3 & l_3 & & & \\ & & \vdots & \vdots & & \\ & & & m_{n-1} & l_{n-1} & \\ & & & & m_n & l_n \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}=\begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix}$$

其中 $y_1=d_1/l_1$，其余各项的递推计算关系是：

$$y_i=(d_i-m_iy_{i-1})/l_i,\ i=2,3,\cdots,n$$

对于第二个方程，求解最终结果 x_i：

$$\begin{bmatrix} 1 & u_1 & & & & \\ & 1 & u_2 & & & \\ & & 1 & u_3 & & \\ & & & \vdots & \vdots & \\ & & & & 1 & u_{n-1} \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}$$

其中 $x_n = y_n$，其余各项的递推求解关系是：

$$x_i = y_i - u_i x_{i+1}, \quad i = n-1, n-2, \cdots, 1$$

递推计算 y_i 和 x_i 的过程分别被形象地形容为“追的过程”和“赶的过程”，这也是追赶法得名的原因，实际上这种方法在国际上叫作托马斯法。在这里需要强调一下，对三角矩阵的克洛脱分解需要满足几个条件，否则无法进行，这几个条件分别是：

(1) $a_i \neq 0, i = 2,3,\cdots,n$

(2) $|a_1| > |c_1|, |a_n| > |b_n|$

(3) $|a_i| > |b_i| + |c_i|, i = 2,3,\cdots,n-1$

下面就给出一个追赶法求解方程组的通用算法实现，在使用之前需要判断系数矩阵是否是对三角矩阵，并且满足上述三个条件，相关的判断请读者自行添加：

```
/*追赶法求对三角矩阵方程组的解*/
bool ThomasEquation::Resolve(std::vector<double>& xValue)
{
    assert(xValue.size() == m_DIM);

    std::vector<double> L(m_DIM);
    std::vector<double> M(m_DIM);
    std::vector<double> U(m_DIM);
    std::vector<double> Y(m_DIM);

    /*消元，追的过程*/
    L[0] = m_matrixA[ARR_INDEX(0, 0, m_DIM)];
    U[0] = m_matrixA[ARR_INDEX(0, 1, m_DIM)] / L[0];
    Y[0] = m_bVal[0] / L[0];
    for(int i = 1; i < m_DIM; i++)
    {
        M[i] = m_matrixA[ARR_INDEX(i, i - 1, m_DIM)];
        L[i] = m_matrixA[ARR_INDEX(i, i, m_DIM)] - M[i] * U[i - 1];
        U[i] = m_matrixA[ARR_INDEX(i - 1, i, m_DIM)] / L[i];
        Y[i] = (m_bVal[i] - M[i] * Y[i - 1]) / L[i];
    }
    /*回代求解，赶的过程*/
    xValue[m_DIM - 1] = Y[m_DIM - 1];
    for(int i = m_DIM - 2; i >= 0; i--)
    {
        xValue[i] = Y[i] - U[i] * xValue[i + 1];
```

```
    }

    return true;
}
```

12.3.6 三次样条曲线拟合算法实现

根据 12.3.4 节对三次样条函数的推导分析，三次样条曲线拟合算法的核心可分为三部分，第一部分是根据推导结果计算关于三次样条函数的“矩”的方程组的系数矩阵，第二部分就是用追赶法求解方程组，得到各个区间的三次样条函数，第三部分就是根据每个拟合点的输入的值 x_i，确定使用哪个区间的三次样条函数，并使用该区间的三次样条函数计算出三次样条插值 y_i，最后得到的一系列（x_i, y_i）组成的曲线就是三次样条拟合曲线。拟合算法也是按照上面的分析，分以下三个步骤计算插值。

第一步是计算系数矩阵，其中 u_0、v_0、d_0 和 d_n 的值需要单独计算，其余的值可以通过式(12-19)递推计算出来。

第二步是将系数矩阵代入 12.3.5 节给出的追赶法通用算法，求出 M_i 的值。求解之前，先证明一下第一步得到系数矩阵是否满足追赶法的条件。首先，主对角线元素的值都是 2，满足 12.3.5 节的条件(1)。其次，由 u_i 和 v_i 的计算条件可知，$|u_i| < 1$，$|v_i| < 1$，这也满足 12.3.5 节的条件(2)。最后，因为 $a_i = 2$，且 u_i 和 v_i 的和是 1，所以 12.3.5 节的条件(3)也得到了满足。由上判断可知，求解三次样条函数的“矩”的系数矩阵满足使用追赶法求解的条件，可以使用追赶法求解。

第三步是计算插值，需要将第二步计算得到的 M_i 代入式(12-15)，并选择合适的子区间样条函数计算出插值点的值。

下面就给出采用三弯矩法实现的三次样条曲线拟合算法，CalcSpline()函数的参数 Xi 和 Yi 是 n 个插值点（包括起点和终点）的值，boundType 是边界条件类型，b1 和 b2 分别是对应的两个边界条件，这个算法支持第一类和第二类边界条件（包括自然边界条件）。内部的矩阵 matrixA 就是按照公式(12-29)构造的 M_i 方程组的系数矩阵，可用于直接用追赶法求解方程组。CalcSpline()函数的大部分代码都是在构造 M_i 方程组的系数矩阵，首先根据边界条件确定 u_n 、v_0 、d_0 和 d_n，其他系数则根据式(12-19)的递推关系，在 for(int i = 1; i < (m_valN - 1); i++)循环中依次计算出来，最后是利用 12.3.5 节给出的追赶法算法求出 M_i。GetValue()函数负责计算给定区间内任意位置的插值，首先根据 x 的值确定使用哪个子区间的样条函数，然后根据式(12-12)和式(12-14)给出的关系计算插值。

```
void SplineFitting::CalcSpline(double *Xi, double *Yi, int n, int boundType, double b1, double b2)
{
    assert((boundType == 1) || (boundType == 2));

    double *matrixA = new double[n * n];
    if(matrixA == NULL)
    {
        return;
```

```
    }
    double *d = new double[n];
    if(d == NULL)
    {
        delete[] matrixA;
        return;
    }

    m_valN = n;
    m_valXi.assign(Xi, Xi + m_valN);
    m_valYi.assign(Yi, Yi + m_valN);
    m_valMi.resize(m_valN);
    memset(matrixA, 0, sizeof(double) * n * n);

    matrixA[ARR_INDEX(0, 0, m_valN)] = 2.0;
    matrixA[ARR_INDEX(m_valN - 1, m_valN - 1, m_valN)] = 2.0;
    if(boundType == 1) /*第一类边界条件*/
    {
        matrixA[ARR_INDEX(0, 1, m_valN)] = 1.0; //v0
        matrixA[ARR_INDEX(m_valN - 1, m_valN - 2, m_valN)] = 1.0; //un
        double h0 = Xi[1] - Xi[0];
        d[0] = 6 * ((Yi[1] - Yi[0]) / h0 - b1) / h0; //d0
        double hn_1 = Xi[m_valN - 1] - Xi[m_valN - 2];
        d[m_valN - 1] = 6 * (b2 - (Yi[m_valN - 1] - Yi[m_valN - 2]) / hn_1) / hn_1; //dn
    }
    else /*第二类边界条件*/
    {
        matrixA[ARR_INDEX(0, 1, m_valN)] = 0.0; //v0
        matrixA[ARR_INDEX(m_valN - 1, m_valN - 2, m_valN)] = 0.0; //un
        d[0] = 2 * b1; //d0
        d[m_valN - 1] = 2 * b2; //dn
    }
    /*计算 ui,vi,di，i = 2,3,...,n-1*/
    for(int i = 1; i < (m_valN - 1); i++)
    {
        double hi_1 = Xi[i] - Xi[i - 1];
        double hi = Xi[i + 1] - Xi[i];
        matrixA[ARR_INDEX(i, i - 1, m_valN)] = hi_1 / (hi_1 + hi); //ui
        matrixA[ARR_INDEX(i, i, m_valN)] = 2.0;
        matrixA[ARR_INDEX(i, i + 1, m_valN)] = 1 - matrixA[ARR_INDEX(i, i - 1, m_valN)]; //vi = 1 -
            ui
        d[i] = 6 * ((Yi[i + 1] - Yi[i]) / hi - (Yi[i] - Yi[i - 1]) / hi_1) / (hi_1 + hi); //di
    }

    ThomasEquation equation(m_valN, matrixA, d);
    equation.Resolve(m_valMi);
    m_bCalcCompleted = true;

    delete[] matrixA;
    delete[] d;
}
double SplineFitting::GetValue(double x)
{
    if(!m_bCalcCompleted)
```

```
    {
        return 0.0;
    }
    if((x < m_valXi[0]) || (x > m_valXi[m_valN - 1]))
    {
        return 0.0;
    }
    int i = 0;
    for(i = 0; i < (m_valN - 1); i++)
    {
        if((x >= m_valXi[i]) && (x < m_valXi[i + 1]))
            break;
    }
    double hi = m_valXi[i + 1] - m_valXi[i];
    double xi_1 = m_valXi[i + 1] - x;
    double xi = x - m_valXi[i];

    double y = xi_1 * xi_1 * xi_1 * m_valMi[i] / (6 * hi);
    y += (xi * xi * xi * m_valMi[i + 1] / (6 * hi));

    double Ai = (m_valYi[i + 1] - m_valYi[i]) / hi - (m_valMi[i + 1] - m_valMi[i]) * hi / 6.0;
    y += Ai * x;
    double Bi = m_valYi[i + 1] - m_valMi[i + 1] * hi * hi / 6.0 - Ai * m_valXi[i + 1];
    y += Bi;
    return y;
}
```

12.3.7　三次样条曲线拟合的效果

本节将用定义一个原始函数，从原始函数的某个区间上抽取 9 个插值点，根据这 9 个插值点和原函数的边界条件，利用三次样条曲线插值进行曲线拟合，并将原始曲线和拟合曲线做对比，展示一下三次样条曲线拟合的效果。

首先定义原始函数：

$$f(x) = \frac{3}{1+x^2}$$

选择区间[0.0, 8.0]上的 9 个点作为插值点，计算各点的值如表 12-2 所示：

表12-2　原函数$f(x)$在各插值点的值

x	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0
y	3.0	1.5	0.6	0.3	0.1765	0.1154	0.0811	0.06	0.0462

求 $f(x)$的导函数 $f'(x)$：

$$f'(x) = \frac{-6x}{(1+x^2)^2}$$

根据 $f'(x)$计算出在区间端点处的两个第一类边界条件 $f'(0.0) = 0.0, f'(8.0) = -0.01136$。利用

表 12-2 中的数据和这两个边界条件，计算出三次样条插值函数，并从 0.0 开始，以 0.01 为步长，连续求 800 个点的插值，将这些点连成曲线得到拟合曲线。为了做对比，同样从 0.0 开始，以 0.01 为步长，用 $f(x)$函数连续计算 800 个点的原值，将这些点连成曲线得到原始曲线。分别用不同的颜色画出这两条曲线，如图 12-2 所示。

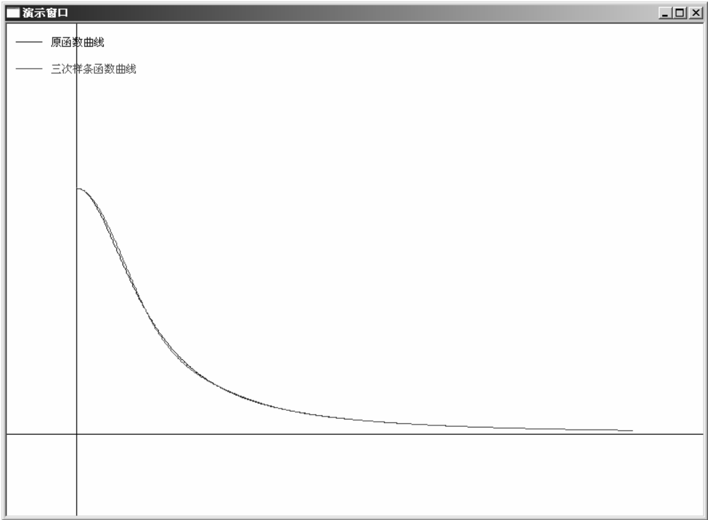


图 12-2　拟合曲线和原始曲线对比

从图 12-2 可以看到，三次样条曲线拟合的效果非常好。同样在[0.0, 8.0]区间上，如果增加插值点的个数，将获得更好的拟合效果。比如以 0.5 为单位，将插值点增加到 17 个，则拟合的曲线与原始曲线几乎完全重合。

12.4　总结

本章介绍了两种常见的曲线拟合算法：最小二乘插值法和三次样条曲线插值法，并通过两个简单的例子介绍了两种算法的应用场景。无论是这两种插值算法，还是求解方程组的高斯消元法和追赶法，都是非常简单的算法，实现也不复杂，但是在现实生活中却到处都有体现。小到一个物理实验，大到工业制造，算法的应用无处不在。但是，它们都很简单，并不神秘，如果你也有这种感觉，本章的目的就达到了。

12.5　参考资料

[1]　李庆杨，关治，白峰杉. 数值计算原理. 北京：清华大学出版社，1999

[2]　李红. 数值分析. 武汉：华中科技大学出版社，2003

第 13 章 非线性方程与牛顿迭代法

一元非线性方程的求解是高等数学研究的重要课题之一，早在 2000 多年前，古巴比伦的数学家就能解一元二次方程了，中国的《九章算术》也有对一元二次方程求解的记载。目前人们普遍认为低阶（5 阶以下）一元非线性方程可以通过求根公式求解，但是高于或等于 5 阶的一元非线性方程不存在求根公式，要精确求解非常困难。对高阶方程，一般采用迭代法近似求解，牛顿迭代法因为方法简单，迭代收敛速度快而被广泛使用。在第 11 章介绍历法算法的时候，你也可以看到牛顿迭代法可用来求解节气和朔日时间。

13.1 非线性方程求解的常用方法

一元非线性方程的常用求解方法有很多，能够精确求解的方法有开平方法、配方法、因式分解法、公式法等，近似求解的方法有作图法以及各种迭代法。开平方法、配方法和因式分解法适用于一元非线性方程中的一些特殊情况，使用范围有限。公式法适用于低阶方程，对于一元二次方程可以使用韦达公式，一元三次方程可以使用卡尔丹公式或盛金公式，公式法比较适合编写计算机算法求解。作图法简单，但是精度不高，可用于使用迭代法时估计迭代初始值。

由迭代法求近似解有很多种方法，有一些迭代法受函数性质的影响，收敛特性不是很好，有些情况下如果初始值选择不当可能会导致迭代不能收敛。本章要介绍的二分逼近法和牛顿迭代法都是计算机程序中常用的两种算法，都具有比较好的收敛速度。此外，公式法因为算法简单，在许多特定领域内的软件中也有广泛的应用。

13.1.1 公式法

一元二次方程的求解是中学数学的内容，对于一元二次方程的一般形式：$ax^2 + bx + c = 0$，可使用韦达公式求解方程的两个实数解，韦达公式可表示为：

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

其中 $\Delta=b^2-4ac$ 是解的判别式，当 $\Delta>0$ 时，方程有两个不相等的实数解，当 $\Delta=0$ 时，方程有两个相等的实数解，当 $\Delta<0$ 时，方程没有实数解。高等数学引入了复数域和虚数单位 $i^2=-1$，当 $\Delta<0$ 时，方程有两个不相等的复数解。

一元三次方程也有求根公式，比如卡尔丹公式和盛金公式。卡尔丹公式已经有四百多年的历史，盛金公式则是由中国数学家范盛金在 20 世纪 80 年代发明的一种方法，比卡尔丹公式简洁实用。在卡尔丹公式出现以后，人们又致力于一元四次方程的求根公式，最终卡尔丹的学生费拉里给出了求解一元四次方程的求根公式。在此后的三百多年时间里，人们苦等的一元五次方程的求根公式始终没有出现，很多著名数学家的尝试也都没有结果。直到 1824 年，挪威数学家阿贝尔证明了五次或五次以上的方程不可能有求根公式（这个结论目前还有争议，因为少数特殊的 5 阶方程被证明有求解公式）。

公式法可以求得精确解，并且根据公式法的推导公式编写计算机算法非常简单，因此这样的算法在很多领域中得到了广泛的使用。

13.1.2 二分逼近法

对于实数域的函数 $f(x)$，如果存在实数 k，使得 $f(k)=0$，则 $x=k$ 就是函数 $f(x)$的零点。如果函数 $f(x)$在是连续函数，且在区间$[a, b]$上是单调函数，只要 $f(a)$和 $f(b)$异号，就说明在区间$[a, b]$内一定有零点，此时就可以使用二分逼近法近似地找到这个零点。假设在上述区间上，$f(a)<0$，$f(b)>0$，则可按照以下过程实施二分逼近法。

(1) 如果 $f((a+b)/2)=0$，则$(a+b)/2$ 就是零点；
(2) 如果 $f((a+b)/2)<0$，则零点在区间$[(a+b)/2, b]$上，令 $a=(a+b)/2$，继续从第(1)步开始判断；
(3) 如果 $f((a+b)/2)>0$，则零点在区间$[a, (a+b)/2]$上，令 $b=(a+b)/2$，继续从第(1)步开始判断。

直接按照 $f((a+b)/2)=0$ 判断是很难的，通常只要 $f((a+b)/2)$在精度允许的范围内逼近 0 时就可以结束二分逼近过程，将$(a+b)/2$ 作为零点，在精度和计算速度二者之间取折衷。除了判断 $f((a+b)/2)$的值，还可以根据区间$[a, b]$的大小确定结束条件，在精度允许的范围内，只要区间范围小于精度阈值，也可以直接取$(a+b)/2$ 作为零点。

从上述过程可以看到，每次运算之后，区间范围就缩小一半，呈现线性收敛速度。二分法的局限性就是不能计算复根和重根，需要借助其手段确定零点所在区间。`DichotomyEquation()`函数就是二分逼近法的算法实现，参数 a 和 b 是求根区间，f 是求根方程。设方程为 $f(x)=2x^2+3.2x-1.8$，求根精度是 `PRECISION =` 0.000000001，在[-0.8, 8.0]区间上求解 $x=0.440967364$，`while` 循环共做了 34 次循环迭代。

```
double DichotomyEquation(double a, double b, FunctionPtr f)
{
    double mid = (a + b) / 2.0;
    while((b - a) > PRECISION)
    {
```

```
            if(f(a) * f(mid) < 0.0)
            {
                b = mid;
            }
            else
            {
                a = mid;
            }
            mid = (a + b) / 2.0;
        }

        return mid;
    }
```

13.2 牛顿迭代法的数学原理

牛顿法迭代法（Newton's method）又称为牛顿–拉弗森方法（Newton-Raphson method），它是一种在实数域和复数域上近似求解方程的方法。方法使用函数 $f(x)$的泰勒级数的前面几项来寻找方程 $f(x)=0$ 的根。

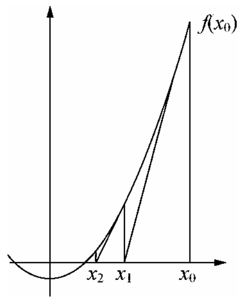


图 13-1 牛顿迭代法逼近示意图

首先，选择一个接近函数 $f(x)$零点的 x_0 作为迭代初始值，计算相应的 $f(x_0)$和切线斜率 $f'(x_0)$（这里 $f'(x)$是函数 $f(x)$的一阶导函数）。然后我们经过点$(x_0,f(x_0))$做一条斜率为 $f'(x_0)$的直线，该直线与 x 轴有一个交点，可通过以下方程的求解得到这个交点的 x 坐标：

$$f(x_0)= (x_0-x)\cdot f'(x_0)$$

求解这个方程，可以得到：

$$x = x_0 - f(x_0)/f'(x_0)$$

我们将新求得的点的 x 坐标命名为 x_1，通常 x_1 会比 x_0 更接近方程 $f(x)=0$ 的解。因此我们现在可以利用 x_1 开始下一轮迭代。根据上述方程中 x_1 和 x_0 的关系，可以得到一个求解 x 的迭代公式：

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

这就是牛顿迭代公式。目前已经证明，如果 $f(x)$的一阶导函数 $f'(x)$是连续函数，并且待求的零点 x 是孤立的，则在零点 x 周围存在一个区间，只要初始值 x_0 位于这个区间，牛顿法迭代必定收敛。并且，只要 $f'(x)\neq 0$，牛顿迭代法将具有平方收敛的性能。这意味着每迭代一次，结果的有效数字将增加一倍，这比二分逼近法的线性收敛速度快了一个数量级。

13.3 用牛顿迭代法求解非线性方程的实例

牛顿迭代法原理简单，求根收敛速度快，编制算法简单，因此在计算机上获得了广泛的应用。牛顿迭代法和其他迭代法一样，使用自变量 x 作为迭代变量，使用牛顿迭代公式建立迭代关系，通过求解的精度控制迭代退出条件。

13.3.1 导函数的求解与近似公式

牛顿迭代公式中需要计算函数的导数，直接根据原函数推出一阶导函数，然后计算导函数的值有点困难，一般都是利用导数的数学原理，使用近似公式直接求函数在某一点的导数。导数的数学定义是：当函数 $y = f(x)$的自变量 x 在一点 x_0 上产生一个增量 Δx 时，函数输出值的增量 Δy 与自变量增量 Δx 的比值在 Δx 趋于 0 时的极限值。如果这个极限值存在，则这个值就是 $f(x)$在 x_0 处的导数，记作 $f'(x_0)$。用公式定义即为：

$$f'(x_0) = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$

极限是在无穷小或无穷大的尺度上考察函数的一些特性，在计算机上无法表达无穷小和无穷大，只能在数据能表达的合法范围内，在满足计算精度的情况下通过最小值来近似模拟。如果无法精确计算导数 $f'(x_0)$，我们仍然采用近似计算方法得到一个满足精度的模拟值。根据导数的数学定义，如果不考虑极限，这个值就是 $\Delta y/\Delta x$ 的值，在 x_0 附近一个非常小的尺度上选择 Δ，可以得到近似的导数值。我们选择按照以下近似公式计算导数值：

$$f'(x_0) = \frac{f(x_0 + 0.000005) - f(x_0 - 0.000005)}{0.00001}$$

计算函数 f 在 x 附近的一阶导数值的算法可定义为：

```
double CalcDerivative(FunctionPtr f, double x)
{
    return (f(x + 0.000005) - f(x - 0.000005)) / 0.00001;
}
```

13.3.2 算法实现

根据牛顿迭代公式，很容易写出牛顿迭代法的算法实现：

```
double NewtonRaphson(FunctionPtr f, double x0)
{
    double x1 = x0 - f(x0) / CalcDerivative(f, x0);
    while(fabs(x1 - x0) > PRECISION)
    {
        x0 = x1;
        x1 = x0 - f(x0) / CalcDerivative(f, x0);
    }
```

```
    return x1;
}
```

参数 x_0 是迭代初始值。选择和 13.1.2 节相同的函数，并将迭代初始值设置为区间最大值 8.0，使用牛顿迭代法也只需要 7 次迭代，就可以得到和二分逼近法精度一样的近似解。选择初始值–8.0 从另一个方向计算，还可以得到另一个解 $x = -2.040967365$，计算这个解也只需要 6 次迭代，可见牛顿迭代法的收敛速度是超线性的。

13.4 参考资料

[1] 同济大学数学系. 高等数学（第六版）. 北京：高等教育出版社，2007

[2] 维基百科：http://zh.wikipedia.org/wiki/牛顿迭代法

[3] 李庆杨，王超能，易大义. 数值分析（第五版）. 北京：清华大学出版社，2008

[4] 李庆杨，关治，白峰杉. 数值计算原理. 北京：清华大学出版社，2000

[5] 李红. 数值分析. 武汉：华中科技大学出版社，2003

第 14 章 计算几何与计算机图形学

我的专业是计算机辅助设计（CAD），算是一半机械一半软件，《计算机图形学》是必修课，也是我最喜欢的课程。热衷于用代码摆平一切的我几乎将这本教科书上的每种算法都实现了一遍，这种重复劳动虽然意义不大，但是收获很多，特别是丢弃了多年的数学又重新捡起来了，算是最大的收获吧。尽管已经毕业多年了，但是每次回顾这些算法的代码，都觉得内心十分澎湃。如果换成现在的我，恐怕再也不会有动力去做这些事情了。

在学习《计算机图形学》之前，总觉得很多东西高深莫测，但实际掌握了之后，却发现其中并无神秘可言，就如同被原始人像神一样崇拜的火却被现代人叼在嘴上玩弄一样的感觉。图形学的基础之一就是计算几何，但是没有理论数学那么高深莫测，它很有实践性，有时候甚至可以简单到让人匪夷所思。计算几何是随着计算机和 CAD 的应用而诞生的一门新兴学科，在国外被称为"计算机辅助几何设计"（Computer Aided Geometric Design，CAGD）。本章就来介绍一些图形学中常见的计算几何算法（顺便晒晒我的旧代码），都是一些图形学中的基础算法，需要一些图形学的知识和数学知识，但是都不难。不信就来看看吧。

14.1 计算几何的基本算法

本节要介绍一些图形学常用的计算几何方法，涉及了向量、点线关系以及点与多边形关系求解等数学知识，还有一些平面几何的基本原理。事先声明一下，文中涉及的算法实现都出于解释原理以及揭示算法实质的目的。在算法效率和可读性二者的考量上，更注重可读性，有时候为了提高可读性，甚至会刻意采取"效率不高"的代码形式。在实际工程中使用的代码肯定更紧凑，更高效，但是算法原理都是一样的，请读者们对此有正确的认识。

14.1.1 点与矩形的关系

计算机图形学和数学到底有什么关系？我们先来看几个例子，增加一些感性的认识。例如判断一个点是否在矩形内的算法，就是一个很简单的算法，但是却非常重要。比如你在一个按钮上点击鼠标，系统如何知道你要触发的是这个按钮而不是另一个按钮对应的事件？这就是一个点是

否在矩形内的判断处理。Windows 的 API 提供了 PtInRect()函数，实现方法其实就是判断点的 x 坐标和 y 坐标是否同时落在矩形的 x 坐标范围和 y 坐标范围内，算法实现也很简单：

```
bool IsPointInRect(const Rect& rc, const Point& p)
{
    double xr = (p.x - rc.p1.x) * (p.x - rc.p2.x);
    double yr = (p.y - rc.p1.y) * (p.y - rc.p2.y);

    return ( (xr <= 0.0) && (yr <= 0.0) );
}
```

看看 IsPointInRect()函数的实现是否和你想象的不一样？由于 IsPointInRect()函数并不假设矩形的两个定点是按照坐标轴升序排列的，所以在算法实现时就考虑了所有可能的坐标范围。有时候硬件实现乘法有困难或受限制于 CPU 乘法指令的效率，工程上通常使用一种避免乘法运算的算法，这种算法虽然代码繁琐了一点，但是非常高效。我在博客中介绍了这种算法，读者可通过我的博客了解到具体的算法实现。

IsPointInRect()函数使用的是平面直角坐标系，如果不特别说明，本文所有的算法都是基于平面直角坐标系设计的。另外，IsPointInRect()函数没有指定特别的浮点数精度范围，默认是系统浮点数的最大精度，只在某些必须要与 0 比较的情况下，采用 10^{-8} 次方精度，如无特别说明，本章所有的算法都这样处理。

14.1.2 点与圆的关系

现在考虑复杂一点，如果图形界面的按钮不是矩形，而是圆形的，该怎么办呢？当然就是判断点是否在圆内部。判断算法的原理就是计算点到圆心的距离 d，然后与圆半径 r 进行比较。若 $d<r$，则说明点在圆内，若 $d=r$，则说明点在圆上，若 $d>r$，则说明点在圆外。这就要提到计算平面上两点距离的算法。以图 14-1 为例，计算平面上任意两点之间的距离主要依据著名的勾股定理，代码如下：

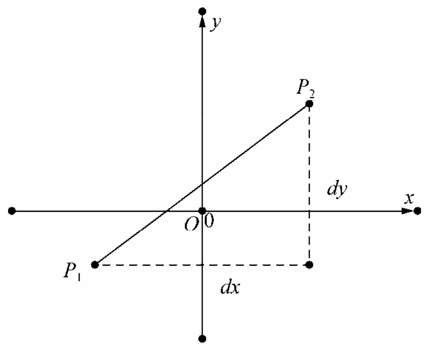


图 14-1 平面两点距离计算示意图

```
double PointDistance(const Point& p1, const Point& p2)
{
    return std::sqrt( (p1.x-p2.x)*(p1.x-p2.x)+ (p1.y-p2.y)*(p1.y-p2.y) );
}
```

14.1.3 矢量的基础知识

现在再考虑复杂一点，如果按钮是个不规则的多边形区域呢？别以为这个考虑没有意义，很多多媒体软件和游戏都用各种形状的不规则图案作为热点（hot spot），Windows 也提供了一个名为 PtInRegion()的 API，用于判断点是否在一个不规则区域中。我们对问题进行简化，就是判断一个点是否在多边形内。判断点 P 是否在多边形内是计算几何中一个非常基本的算法，最常用的

方法是射线法。以 P 点为端点，向左方做射线 L，然后沿着 L 从无穷远处开始向 P 点移动，当遇到多边形的某一条边时，记为与多边形的第一个交点，表示进入多边形内部，继续移动，当遇到另一个交点时，表示离开多边形内部。由此可知，当 L 与多边形的交点个数是偶数时，表示 P 点在多边形外，当 L 与多边形交点个数是奇数时，表示 P 点在多边形内部。

由此可见，要实现判断点是否在多边形内的算法，需要知道直线段求交算法，而求交算法又涉及矢量的一些基本概念，因此在实现这个算法之前，先讲一下矢量的基本概念与算法。

1. 什么是矢量

什么是矢量？简单地讲，就是既有大小又有方向的量，在数学中又常被称为向量。矢量有几何表示、代数表示和坐标表示等多种表现形式，本节讨论的是几何表示。如果一条线段的端点是有次序之分的，我们把这种线段称为有向线段（directed segment），比如线段 P_1P_2，如果起始端点 P_1 就是坐标原点(0, 0)，P_2 的坐标是(x, y)，则线段 P_1P_2 的二维矢量坐标表示就是 $P = (x, y)$。

2. 矢量的加法与减法

现在来看几个与矢量有关的重要概念，首先是矢量的加减法。假设有二维矢量 $P=(x_1, y_1)$，$Q=(x_2, y_2)$，则矢量加法定义为：

$$P + Q = (x_1 + x_2 , y_1 + y_2) \tag{14-1}$$

同样地，矢量减法定义为：

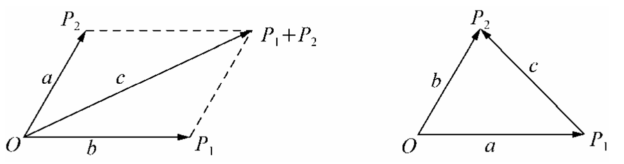
$$P - Q = (x_1 - x_2 , y_1 - y_2) \tag{14-2}$$

根据矢量加减法的定义，矢量的加减法满足以下性质：

$$P + Q = Q + P$$

$$P - Q = -(Q - P)$$

图 14-2 展示了矢量加法和减法的几何意义，由于几何中直线段的两个点不可能刚好在原点，因此线段 P_1P_2 的矢量其实就是 $OP_2 - OP_1$ 的结果，如图 14-2b 所示。



(a) 矢量加法的几何表示　(b) 矢量减法的几何表示

图 14-2　矢量加法和矢量减法的几何意义

3. 矢量的叉积

另一个比较重要的概念是矢量的叉积（外积）。计算矢量的叉积是判断直线和线段、线段和线段以及线段和点的位置关系的核心算法。假设有二维矢量 $P = (x_1, y_1)$，$Q = (x_2 , y_2)$，则矢量的

叉积定义为：

$$P \times Q = x_1{*}y_2 - x_2{*}y_1 \tag{14-3}$$

向量叉积的几何意义可以描述为由坐标原点(0,0)、P、Q 和 $P + Q$ 所组成的平行四边形的面积，而且是个带符号的面积，由此可知，矢量的叉积具有以下性质：

$$P \times Q = -(Q \times P)$$

叉积的结果 $P \times Q$ 是 P 和 Q 所在平面的法矢量，它的方向是垂直与 P 和 Q 所在的平面，并且按照 P、Q 和 $P \times Q$ 的次序构成右手系，所以叉积的另一个非常重要性质是可以通过它的符号可以判断两矢量相互之间位置关系是顺时针还是逆时针关系，具体说明如下。

(1) 如果 $P \times Q > 0$， 则 Q 在 P 的逆时针方向；
(2) 如果 $P \times Q < 0$， 则 Q 在 P 的顺时针方向；
(3) 如果 $P \times Q = 0$， 则 Q 与 P 共线（但可能方向是反的）。

给定向量 $P = (x_1, y_1)$，$Q = (x_2, y_2)$，计算叉积的算法实现为：

```
double CrossProduct(double x1, double y1, double x2, double y2)
{
    return x1 * y2 - x2 * y1;
}
```

4. 矢量的点积

最后要介绍的概念是矢量的点积（内积）。假设有二维矢量 $P = (x_1, y_1)$，$Q = (x_2, y_2)$，则矢量的点积定义为：

$$P \cdot Q = x_1{*}x_2 + y_1{*}y_2 \tag{14-4}$$

向量点积的结果是一个标量，它的代数表示是：

$$P \cdot Q = |P|\,|Q| \cos(P, Q) \tag{14-5}$$

(P, Q) 表示向量 P 和 Q 的夹角，如果 P 和 Q 不共线，则根据上式可以得到向量点积的一个非常重要的性质，具体说明如下。

(1) 如果 $P \cdot Q > 0$，则 P 和 Q 的夹角是钝角（大于 90 度）；
(2) 如果 $P \cdot Q < 0$，则 P 和 Q 的夹角是锐角（小于 90 度）；
(3) 如果 $P \cdot Q = 0$，则 P 和 Q 的夹角是 90 度。

给定向量 $P = (x_1, y_1)$，$Q = (x_2, y_2)$，计算点积的算法实现为：

```
double DotProduct(double x1, double y1, double x2, double y2)
{
    return x1 * x2 + y1 * y2;
}
```

了解了矢量的概念以及矢量的各种运算的几何意义和代数意义后，就可以开始解决各种计算

几何的简单问题了，回想本节开始提到的点与多边形的关系问题，首先要解决的就是判断点和直线段的位置关系问题。

14.1.4 点与直线的关系

根据矢量叉积的几何意义，如果线段所表示的矢量和点的矢量的叉积是 0，就说明点在线段所在的直线上，相对于坐标原点 O 来说，线段的矢量其实就是线段终点 $P_2=[x_2, y_2]$的矢量 OP_2 减线段起点 $P_1=[x_1, y_1]$的矢量 OP_1 的结果，因此线段 P_1P_2 的矢量可以表示为 $P_1P_2 = (x_2-x_1, y_2-y_1)$。如果要判断点 P 是否在线段 P_1P_2 上，就要判断矢量 P_1P_2 和矢量 OP 的叉积是否是 0。需要注意的是，叉积为 0 只能说明点 P 与线段 P_1P_2 所在的直线共线，并不能说明点 P 一定会落在 P_1P_2 区间上，因此只是一个必要条件。要正确判断 P 在线段 P_1P_2 上，还需要做一个排斥试验，就是检查点 P 是否在以直线段为对角线的矩形空间内，如果以上两个条件都为真，即可判定点在线段上。有了上述原理，算法实现就比较简单了，以下就是判断点是否在线段上的算法：

```
bool IsPointOnLineSegment(const LineSeg& ls, const Point& pt)
{
    Rect rc;

    GetLineSegmentRect(ls, rc);
    double cp = CrossProduct(ls.pe.x - ls.ps.x, ls.pe.y - ls.ps.y,
                             pt.x - ls.ps.x, pt.y - ls.ps.y); //计算叉积

    return ( (IsPointInRect(rc, pt)) //排除实验
           && IsZeroFloatValue(cp) ); //1E-8 精度
}
```

GetLineSegmentRect()函数获取直线的矩形包围盒，为排斥试验做准备。

14.1.5 直线与直线的关系

矢量叉积计算在计算几何中的另一个用途是直线段求交。求交算法是计算机图形学的核心算法，也是体现速度和稳定性的重要标志，高效并且稳定的求交算法是任何一个 CAD 软件都必需要重点关注的。求交包含两层概念，一个是判断是否相交，另一个是求出交点。直线（段）的求交算法相对来说是比较简单的，首先来看看如何判断两直线段是否相交。

常规的代数计算通常分三步，首先根据线段还原出两条线段所在直线的方程，然后联立方程组求出交点，最后再判断交点是否在线段区间上。常规的代数方法非常繁琐，每次都要解方程组求交点，特别是交点不在线段区间的情况，计算交点就是做无用功。计算几何方法判断直线段是否有交点通常分两个步骤完成，这两个步骤分别是快速排斥试验和跨立试验。举个例子，要判断线段 P_1P_2 和线段 Q_1Q_2 是否有交点，则需要做以下两个步骤。

(1) 快速排斥试验

设以线段 P_1P_2 为对角线的矩形为 R_1， 设以线段 Q_1Q_2 为对角线的矩形为 R_2，如果 R_1 和 R_2

不相交，则两线段不会有交点。

(2) 跨立试验

如果两线段相交，则两线段必然相互跨立对方。所谓跨立，指的是一条线段的两端点分别位于另一线段所在直线的两边。判断是否跨立，还是要用到矢量叉积的几何意义。以图 14-3 所示内容为例，若 P_1P_2 跨立 Q_1Q_2，则矢量(P_1-Q_1)和(P_2-Q_1)位于矢量(Q_2-Q_1)的两侧，即：

$$(P_1-Q_1)\times(Q_2-Q_1) * (P_2-Q_1)\times(Q_2-Q_1) < 0$$

上式可改写成：

$$(P_1-Q_1)\times(Q_2-Q_1) * (Q_2-Q_1)\times(P_2-Q_1) > 0$$

当$(P_1-Q_1) \times (Q_2-Q_1) = 0$ 时，说明线段 P_1P_2 和 Q_1Q_2 共线（但是不一定有交点）。同理判断 Q_1Q_2 跨立 P_1P_2 的依据是：

$$(Q_1-P_1)\times(P_2-P_1) * (Q_2-P_1)\times(P_2-P_1) < 0$$

具体情况如图 14-3 所示：

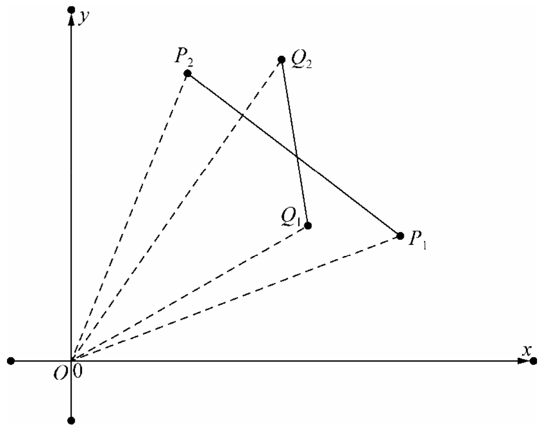


图 14-3　直线段跨立试验示意图

根据矢量叉积的几何意义，跨立试验只能证明线段的两端点位于另一个线段所在直线的两边，但是不能保证是在另一直线段的两端，因此，跨立试验只是证明两条线段有交点的必要条件，必需和快速排斥试验一起才能组成直线段相交的充分必要条件。根据以上分析，两条线段有交点的完整判断依据就是：以两条线段为对角线的两个矩形有交集；两条线段相互跨立。

判断直线段跨立用计算叉积算法的 CrossProduct()函数即可，还需要一个判断两个矩形是否有交的算法。矩形求交也是最简单的求交算法之一，原理就是根据两个矩形的最大、最小坐标判断。所谓的最大最小坐标判断，就是在 x 坐标方向和 y 坐标方向分别满足最大值最小值法则，简单解释这个法则就是每个矩形在每个方向上的坐标最大值都要大于另一个矩形在这个坐标方向

上的坐标最小值，否则在这个方向上就不能保证一定有位置重叠。由以上分析，判断两个矩形是否相交的算法就可以如下实现：

```
bool IsRectIntersect(const Rect& rc1, const Rect& rc2)
{
    return ( (std::max(rc1.p1.x, rc1.p2.x) >= std::min(rc2.p1.x, rc2.p2.x))
        && (std::max(rc2.p1.x, rc2.p2.x) >= std::min(rc1.p1.x, rc1.p2.x))
        && (std::max(rc1.p1.y, rc1.p2.y) >= std::min(rc2.p1.y, rc2.p2.y))
        && (std::max(rc2.p1.y, rc2.p2.y) >= std::min(rc1.p1.y, rc1.p2.y)) );
}
```

完成了排斥试验和跨立试验的算法，最后判断直线段是否有交点的算法就水到渠成了：

```
bool IsLineSegmentIntersect(const LineSeg& ls1, const LineSeg& ls2)
{
    if(IsLineSegmentExclusive(ls1, ls2)) //排斥实验
    {
        return false;
    }
    //( P1 - Q1 ) ×( Q2 - Q1 )
    double p1xq = CrossProduct(ls1.ps.x - ls2.ps.x, ls1.ps.y - ls2.ps.y,
                               ls2.pe.x - ls2.ps.x, ls2.pe.y - ls2.ps.y);
    //( P2 - Q1 ) ×( Q2 - Q1 )
    double p2xq = CrossProduct(ls1.pe.x - ls2.ps.x, ls1.pe.y - ls2.ps.y,
                               ls2.pe.x - ls2.ps.x, ls2.pe.y - ls2.ps.y);

    //( Q1 - P1 ) ×( P2 - P1 )
    double q1xp = CrossProduct(ls2.ps.x - ls1.ps.x, ls2.ps.y - ls1.ps.y,
                               ls1.pe.x - ls1.ps.x, ls1.pe.y - ls1.ps.y);
    //( Q2 - P1 ) ×( P2 - P1 )
    double q2xp = CrossProduct(ls2.pe.x - ls1.ps.x, ls2.pe.y - ls1.ps.y,
                               ls1.pe.x - ls1.ps.x, ls1.pe.y - ls1.ps.y);

    //跨立实验
return ( (p1xq * p2xq <= 0.0) && (q1xp * q2xp <= 0.0) );
}
```

IsLineSegmentExclusive()函数就是调用 IsRectIntersect()函数根据结果做排斥判断，此处不再列出代码。

14.1.6　点与多边形的关系

好了，现在我们已经了解了矢量叉积的意义，以及判断直线段是否有交点的算法，现在回过头看看文章开始部分的讨论的问题：如何判断一个点是否在多边形内部？根据射线法的描述，其核心是求解从 P 点发出的射线与多边形的边是否有交点。注意，这里说的是射线，而我们前面讨论的都是线段，好像不适用吧？没错，确实不适用，但是我要介绍一种用计算机解决问题时常用的建模思想，应用了这种思想之后，我们前面讨论的方法就适用了。什么思想呢？就是根据问题域的规模和性质抽象和简化模型的思想，这可不是故弄玄虚，说说具体的思路吧。

计算机是不能表示无穷大和无穷小，计算机处理的每一个数都有确定的值，而且必须有确定

的值。我们面临的问题域是整个实数空间的坐标系，在每个维度上都是从负无穷到正无穷，比如射线，就是从坐标系中一个明确的点到无穷远处的连线。这就有点为难计算机了，为此我们需要简化问题的规模。假设问题中多边形的每个点的坐标都不会超过(–10000.0, +10000.0)区间（比如我们常见的图形输出设备都有大小的限制），我们就可以将问题域简化为(–10000.0, +10000.0)区间内的一小块区域，对于这块区域来说，≥10000.0 就意味着无穷远。你肯定已经明白了，数学模型经过简化后，算法中提到的射线就可以理解为从模型边界到内部点 P 之间的线段，前面讨论的关于线段的算法就可以使用了。

射线法的基本原理是判断由 P 点发出的射线与多边形的交点个数，交点个数是奇数表示 P 点在多边形内（在多边形的边上也视为在多边形内部的特殊情况），正常情况下经过点 P 的射线应该如图 14-4a 所示，但是也可能碰到多种非正常情况，比如刚好经过多边形一个定点的情况，如图 14-4b，这会被误认为和两条边都有交点，还可能与某一条边共线如图 14-4c 和 14-4d，共线就有无穷多的交点，导致判断规则失效。还要考虑凹多边形的情况，如图 14-4e 所示。

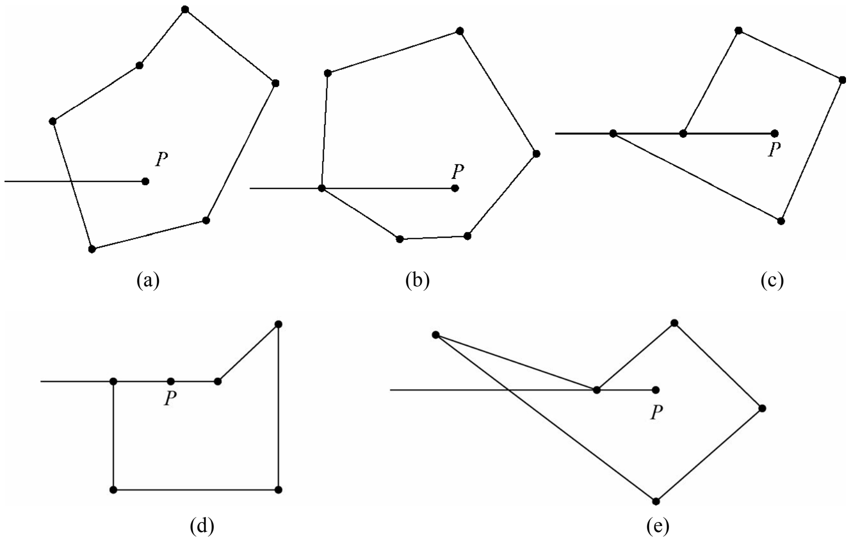


图 14-4　射线法可能遇到的各种交点情况

针对这些特殊情况，在对多边形的每条边进行判断时，要考虑以下这些特殊情况，假设当前处理的边是 P_1P_2，则有以下原则。

(1) 如果点 P 在边 P_1P_2 上，则直接判定点 P 在多边形内；

(2) 如果从 P 发出的射线正好穿过 P_1 或者 P_2，那么这个交点会被算作 2 次（因为在处理以 P_1 或 P_2 为端点的其它边时可能已经计算过这个点了），对这种情况的处理原则是：如果 P 的 y 坐标与 P_1、P_2 中较小的 y 坐标相同，则忽略这个交点；

(3) 如果从 P 发出的射线与 P_1P_2 平行，则忽略这条边。

对于原则(3)，需要判断两条直线是否平行，通常的方法是计算两条直线的斜率，但是本算法因为只涉及直线段（射线也被模型简化为长线段了），就简化了很多，判断直线是否水平，只要比较一下线段起始点的 y 坐标是否相等就行了，而判断直线是否垂直，也只要比较一下线段起始点的 x 坐标是否相等就行了。

应用以上原则后，扫描线法判断点是否在多边形内的算法流程就完整了，图 14-5 就是算法的流程图。

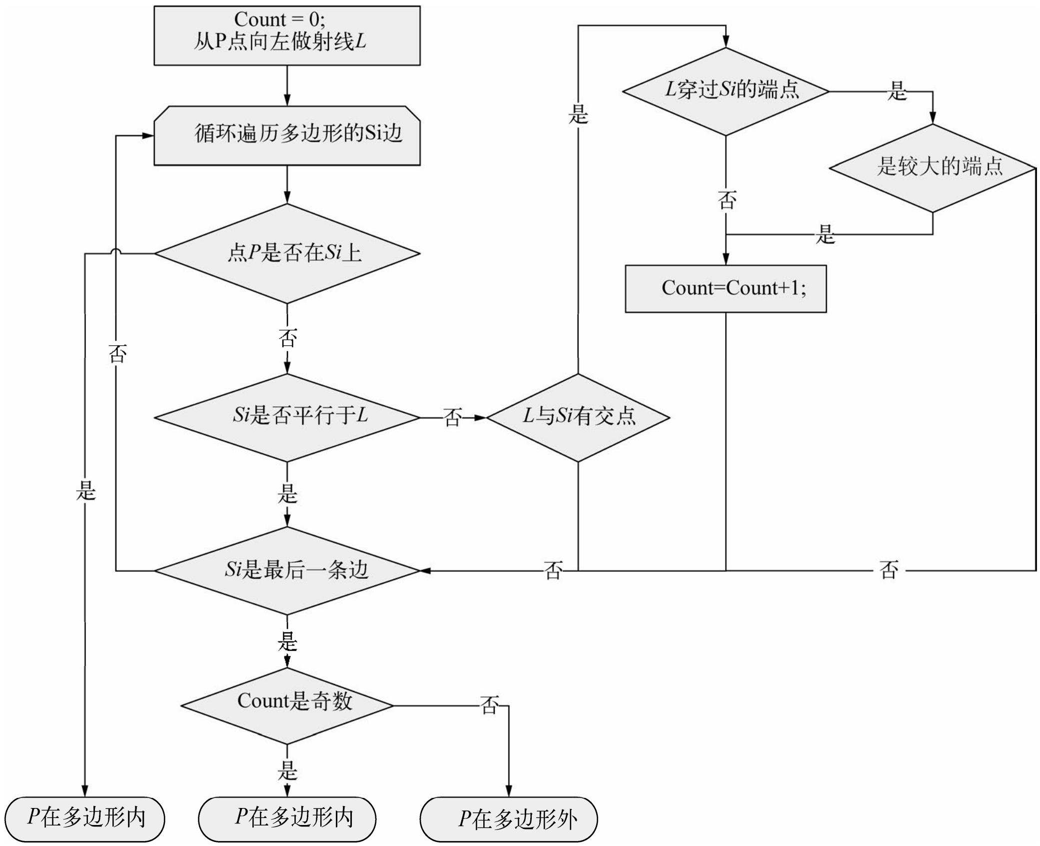


图 14-5　判断点是否在多边形内的扫描线算法流程

有了流程图做指导，算法实现就水到渠成了：

```
bool IsPointInPolygon(const Polygon& py, const Point& pt)
{
    assert(py.IsValid()); /*只考虑正常的多边形*/

    int count = 0;
    LineSeg ll = LineSeg(pt, Point(INFINITE, pt.y)); /*射线 L*/
    for(int i = 0; i < py.GetPolyCount(); i++)
```

```
    {
        /*当前点和下一个点组成线段 P1P2*/
        LineSeg pp = LineSeg(py.pts[i], py.pts[(i + 1) % py.GetPolyCount()]);
        if(IsPointOnLineSegment(pp, pt))
        {
            return true;
        }

        if(!pp.IsHorizontal())
        {
            if((IsSameFloatValue(pp.ps.y, pt.y)) && (pp.ps.y > pp.pe.y))
            {
                count++;
            }
            else if((IsSameFloatValue(pp.pe.y, pt.y)) && (pp.pe.y > pp.ps.y))
            {
                count++;
            }
            else
            {
                if(IsLineSegmentIntersect(pp, ll))
                {
                    count++;
                }
            }
        }
    }

    return ((count % 2) == 1);
}
```

在图形学领域实施的真正工程代码，通常还会增加一个多边形的外包矩形快速判断，对点根本就不在多边形周围的情况做快速排除，提高算法效率。这又涉及求多边形外包矩形的算法，这个算法也很简单，就是遍历多边形的所有节点，找出各个坐标方向上的最大最小值。在本章的配套代码中有这个算法的实现，读者也可以自己完成这个算法。

除了扫描线法，还可以通过多边形边的法矢量方向、多边形面积以及角度和等方法判断点与多边形的关系。但是这些算法要么只支持凸多边形，要么需要复杂的三角函数运算（多边形边数小于 44 时，可采用近似公式计算夹角和，避免三角函数运算），使用的范围有限，只有扫描线法被广泛应用。

14.2 直线生成算法

在欧氏几何空间中，平面方程就是一个三元一次方程，直线就是两个非平行平面的交线，所以直线方程就是两个三元一次方程组联立。但是在平面解析几何中，直线的方程就简单得多了。平面几何中直线方程有多种形式，一般式直线方程可用于描述所有直线：

$$Ax+By+C=0 \quad（A、B\text{ 不同时为 }0） \tag{14-6}$$

当知道直线上一点坐标(X_0,Y_0)和直线的斜率 K 存在时，可以用点斜式方程：

$$Y - Y_0 = K(X - X_0)\text{（当 }K\text{ 不存在时，直线方程简化成 }X = X_0\text{）} \tag{14-7}$$

当知道直线上的两个点(X_0,Y_0)和(X_1,Y_1)时，还可以用两点式方程描述直线：

$$\frac{Y - Y_0}{Y_1 - Y_0} = \frac{X - X_0}{X_1 - X_0} \tag{14-8}$$

除了这三种形式的直线方程外，直线方程还有截距式、斜截式等多种形式。在计算机中如何展示直线图形是计算机图形学的重要内容，这就是本节要介绍的直线生成算法。要理解直线生成算法，首先从理解光栅图形与矢量图形的区别，来看看什么是光栅图形扫描转换。

14.2.1　什么是光栅图形扫描转换

在数学范畴内的直线是由没有宽度的点组成的集合，但是在计算机图形学的范畴内，所有的图形包括直线都是输出或显示在点阵设备上的，被称为点阵图形或光栅图形。以显示器为例，现实中常见的显示器（包括 CRT 显示器和液晶显示器）都可以看成由各种颜色和灰度值的像素点组成的像素矩阵，这些点是有大小的，而且位置固定，因此只能近似的显示各种图形。图 14-6 就是对这种情况的一种夸张的放大。

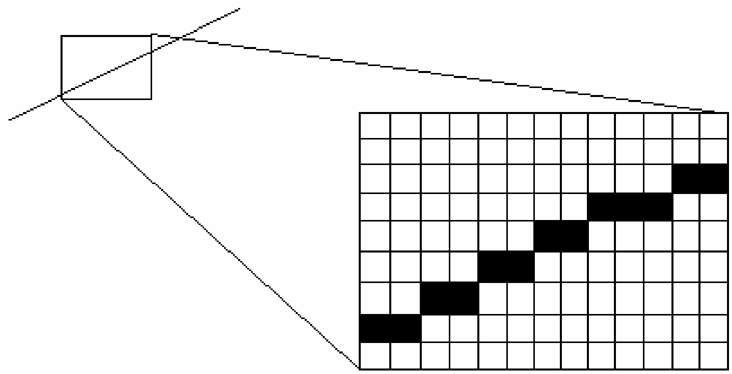


图 14-6　直线在点阵设备上的表现形式

计算机图形学中的直线生成算法，其实包含了两层意思，一层是在解析几何空间中根据坐标构造出平面直线，另一层就是在光栅显示器之类的点阵设备上输出一个最逼近于图形的像素直线，而这就是常说的光栅图形扫描转换。本节就介绍几种常见的直线生成的光栅扫描转换算法，包括数值微分法、Bresenham 算法、对称直线生成算法以及两步算法。

14.2.2　数值微分法

数值微分法（DDA 法）是直线生成算法中最简单的一种，它是一种单步直线生成算法。它

的算法是这样的：首先根据直线的斜率确定是以 X 方向步进还是以 Y 方向步进，然后沿着步进方向每步进一个点（像素），就沿着另一个坐标方向步进 k 个点，k 是直线的斜率，不一定是整数，需要在这个坐标方向对步进后的结果进行圆整。

具体算法的实现，除了判断是按照 X 方向还是按照 Y 方向步进之外，还要考虑直线的方向，也就是起点和终点的关系。下面就是一个支持任意直线方向的数值微分画线算法实例：

```
void DDA_Line(int x1, int y1, int x2, int y2)
{
    double k,dx,dy,x,y,xend,yend;

    dx = x2 - x1;
    dy = y2 - y1;
    if(fabs(dx) >= fabs(dy))
    {
        k = dy / dx;
        if(dx > 0)
        {
            x = x1;
            y = y1;
            xend = x2;
        }
        else
        {
            x = x2;
            y = y2;
            xend = x1;
        }
        while(x <= xend)
        {
            SetDevicePixel((int)x, ROUND_INT(y));
            y = y + k;
            x = x + 1;
        }

    }
    else
    {
        k = dx / dy;
        if(dy > 0)
        {
            x = x1;
            y = y1;
            yend = y2;
        }
        else
        {
            x = x2;
            y = y2;
            yend = y1;
        }
        while(y <= yend)
```

```
        {
            SetDevicePixel(ROUND_INT(x), (int)y);
            x = x + k;
            y = y + 1;
        }
    }
}
```

数值微分法产生的直线比较精确，而且逻辑简单，易于用硬件实现，但是步进量 x、y 和 k 必须用浮点数表示，每一步都要对 x 或 y 进行四舍五入后取整，不利于光栅化或点阵输出。

14.2.3　Bresenham 算法

Bresenham 算法是由 Bresenham 在 1965 年提出的一种单步直线生成算法，是计算机图形学领域使用最广泛的直线扫描转换算法。Bresenham 算法的基本原理就是将光栅设备的各行各列像素中心连接起来构造一组虚拟网格线。按直线从起点到终点的顺序计算直线与各垂直方向网格线的交点，然后确定该列像素中与此交点最近的像素。

图 14-7 就展示了这样一组网格线，每个交点就代表点阵设备上的一个像素点，现在就以图 14-7 为例介绍一下 Bresenham 算法。当算法从一个点(X_i,Y_i)沿着 X 方向向前步进到 X_{i+1} 时，Y 方向的下一个位置只可能是 Y_i 和 Y_{i+1} 两种情况，到底是 Y_i 还是 Y_{i+1} 取决于它们与精确值 y 的距离 d_1 和 d_2 哪个更小。

$$d_1 = y - Y_i \tag{14-9}$$

$$d_2 = Y_{i+1} - y \tag{14-10}$$

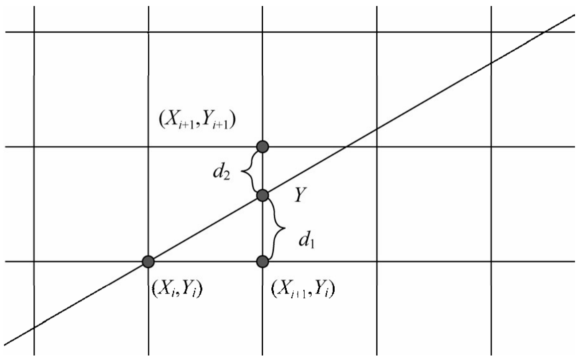


图 14-7　直线 Bresenham 算法示意图

当 $d_1 - d_2 > 0$ 时，Y 方向的下一个位置将是 Y_{i+1}，否则就是 Y_i。由此可见，Bresenham 算法其实和数值微分算法原理是一样的，差别在于 Bresenham 算法中确定 Y 方向下一个点的位置的判断条件的计算方式不一样。现在就来分析一下这个判断条件的计算方法，已知直线的斜率 k 和在 y 轴的截距 b，可推导出 X_{i+1} 位置的精确值 y 如下：

$$y = k X_{i+1} + b \tag{14-11}$$

将式(14-9)、式(14-10)和式(14-11)带入 $d_1–d_2$，可得到式(14-12)：

$$d_1 - d_2 = 2k X_{i+1} - Y_i - Y_{i+1} + 2b \tag{14-12}$$

又因为根据图 14-7 条件，$k = dy / dx$，$Y_{i+1} = Y_i + 1$，$X_{i+1} = X_i + 1$，将此三个关系带入式(14-12)，同时在等式两边乘以 dx，整理后可得到式(14-13)：

$$dx(d_1 - d_2) = 2dyX_i + 2dy - 2dxY_i + dx(2b - 1) \tag{14-13}$$

设 $P_i = dx(d_1 - d_2)$，则：

$$P_i = 2dyX_i + 2dy - 2dxY_i + dx(2b - 1) \tag{14-14}$$

因为图 14-7 的示例中 dx 是大于 0 的值，因此 P_i 的符号与$(d_1- d_2)$一致，现在将初始条件带入可得到最初的第一个判断条件 P_1：

$$P_1 = 2dy - dx$$

根据 X_{i+1} 与 X_i，以及 Y_{i+1} 与 Y_i 的关系，可以推出 P_i 的递推关系：

$$P_{i+1} = P_i + 2dy - 2dx(y_{i+1} - y_i) \tag{14-15}$$

由于 y_{i+1} 可能是 y_i，也可能是 $y_i + 1$，因此，P_{i+1} 就可能是以下两种可能，并且和 y_i 的取值是对应的：

$$P_{i+1} = P_i + 2dy \quad （Y\text{方向保持原值}） \tag{14-16}$$

或

$$P_{i+1} = P_i + 2(dy - dx) \quad （Y\text{方向向前步进 }1） \tag{14-17}$$

根据上面的推导，当 $x_2 > x_1$，$y_2 > y_1$ 时 Bresenham 直线生成算法的计算过程如下。

(1) 画点(x_1, y_1)；计算误差初值 $P_1 = 2dy - dx$；

(2) 求直线的下一点位置：$X_{i+1} = X_i + 1$，如果 $P_i > 0$，则 $Y_{i+1} = Y_i + 1$，否则 $Y_{i+1} = Y_i$，画点(X_{i+1}, Y_{i+1})；

(3) 求下一个误差 P_{i+1}。如果 $P_i > 0$，则 $P_{i+1} = P_i+2(dy - dx)$，否则 $P_{i+1} = P_i+2dy$；

(4) 如果没有结束，则转到步骤(2)，否则结束算法。

下面就给出针对上面推导出的算法源代码（只支持 $x_2 \geqslant x_1$，$y_2 \geqslant y_1$ 的情况）：

```
void Bresenham_Line(int x1, int y1, int x2, int y2)
{
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int p = 2 * dy - dx;
    int x = x1;
    int y = y1;
```

```
        while(x <= x2)
        {
            SetDevicePixel(x, y);
            x++;
            if(p<0)
                p += 2 * dy;
            else
            {
                p += 2 * (dy - dx);
                y += 1;
            }
        }
    }
```

上面的代码只支持一个方向的直线绘制，真正实用的代码要支持各种方向的直线生成，这就要考虑斜率为负值的情况以及 $x_1 > x_2$ 的情况。要支持各种方向的直线生成其实也很简单，就是通过坐标交换，使之符合上面演示算法的要求即可。我在博客中给出了一个实用的 Bresenham 直线生成算法，读者可通过我的博客获得这个算法的实现代码。

Bresenham 算法只实用整数计算，少量的乘法运算都可以通过移位来避免，因此计算量少，效率高。

14.2.4 对称直线生成算法

直线段有个特性，那就是直线段相对于中心点是两边对称的。因此可以利用这个对称性，对其他单步直线生成算法进行改进，使得每进行一次判断或相关计算可以生成相对于直线中点的两个对称点。如此以来，直线就由两端向中间生成。从理论上讲，这个改进可以应用于任何一种单步直线生成算法，本例就只是对 Bresenham 算法进行改进。

改进主要集中在以下几点，首先是循环区间，由$[x_1, x_2]$修改成$[x_1, half]$，half 是区间$[x_1, x_2]$的中点。其次是 X 轴的步进方向改成双向，最后是 Y 方向的值要对称修改，除此之外，算法整体结构不变，下面就是改进后的代码：

```
    void Sym_Bresenham_Line(int x1, int y1, int x2, int y2)
    {
        int dx,dy,p,const1,const2,xs,ys,xe,ye,half,inc;

        int steep = (abs(y2 - y1) > abs(x2 - x1)) ? 1 : 0;
        if(steep == 1)
        {
            SwapInt(&x1, &y1);
            SwapInt(&x2, &y2);
        }
        if(x1 > x2)
        {
            SwapInt(&x1, &x2);
            SwapInt(&y1, &y2);
        }
        dx = x2 - x1;
```

```
        dy = abs(y2 - y1);
        p = 2 * dy - dx;
        const1 = 2 * dy;
        const2 = 2 * (dy - dx);
        xs = x1;
        ys = y1;
        xe = x2;
        ye = y2;
        half = (dx + 1) / 2;
        inc = (y1 < y2) ? 1 : -1;
        while(xs <= half)
        {
            if(steep == 1)
            {
                SetDevicePixel(ys, xs);
                SetDevicePixel(ye, xe);
            }
            else
            {
                SetDevicePixel(xs, ys);
                SetDevicePixel(xe, ye);
            }
            xs++;
            xe--;
            if(p<0)
                p += const1;
            else
            {
                p += const2;
                ys += inc;
                ye -= inc;
            }
        }
    }
```

14.2.5 两步算法

*两步算法*是在生成直线的过程中，每次判断都生成两个点的直线生成算法。上一节介绍的对称直线生成方法也是每次生成两个点，但是它和两步算法的区别就是对称方法的计算和判断是从线段的两端向中点进行，而两步算法是沿着一个方向，一次生成两个点。

当斜率 k 满足条件 $0 \leqslant k < 1$ 时，假如当前点 P 已经确定，如图 14-8 所示，则 P 之后的连续两个点只可能是四种情况：AB，AC，DC 和 DE，两步算法设立决策量 e 作为判断标志，e 的初始值是 $4dy-dx$，其中：

$$dy = y_2 - y_1$$

$$dx = x_2 - x_1$$

为简单起见，先考虑 $dy > dx > 0$ 这种情况。当 $e > 2dx$ 时，P 后两个点将会是 DE 组合，此时 e 的增量是 $4dy-4dx$。当 $dx < e < 2dx$ 时，P 后的两个点将会是 DC 组合，此时 e 的增量是 $4dy$ －

$2dx$。当 $0 < e < dx$ 时，P 后的两个点将会是 AC 组合，此时 e 的增量是 $4dy - 2dx$。当 $e < 0$ 时，P 后的两个点将会是 AB 组合，此时 e 的增量是 $4dy$。综合以上描述，当斜率 k 满足条件 $0 \leqslant k < 1$，且 $dy > dx > 0$ 这种情况下，两步算法可以这样实现：

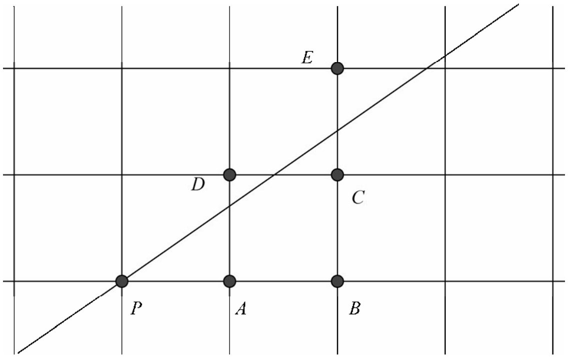


图 14-8　直线两步算法示意图

```
void Double_Step_Line(int x1, int y1, int x2, int y2)
{
    int dx = x2 - x1;
    int dy = y2 - y1;
    int e = dy * 4 - dx;
    int x = x1;
    int y = y1;

    SetDevicePixel(x, y);

    while(x < x2)
    {
        if (e > dx)
        {
            if (e > ( 2 * dx))
            {
                e += 4 * (dy - dx);
                x++;
                y++;
                SetDevicePixel(x, y);
                x++;
                y++;
                SetDevicePixel(x, y);
            }
            else
            {
                e += (4 *dy - 2 * dx);
                x++;
                y++;
                SetDevicePixel(x, y);
                x++;
                SetDevicePixel(x, y);
            }
```

```
            }
            else
            {
                if (e > 0)
                {
                    e += (4 * dy - 2 * dx);
                    x++;
                    SetDevicePixel(x, y);
                    x++;
                    y++;
                    SetDevicePixel(x, y);
                }
                else
                {
                    x++;
                    SetDevicePixel(x, y);
                    x++;
                    SetDevicePixel(x, y);
                    e += 4 * dy;
                }
            }
        }
    }
```

以上函数除了只支持一个方向的直线生成之外，还有其他不完善的地方，比如没有判断最后一个点是否会越界，大量出现的乘法计算可以用移位处理等。仿照 Bresenham 算法一节介绍的方法，很容易将其扩展为支持 8 个方向的直线生成，有兴趣的读者可自己研究实现算法。

14.2.6 其他直线生成算法

除了以上介绍的几种直线生成算法，还有很多其他直线光栅扫描转换算法，比如三步算法、四步算法、中点划线法等，还有人将三步算法结合前面介绍的对称法提出了一种可以一次画六个点的直线生成算法，这里就不多介绍了，有兴趣的读者可以找计算机图形学的相关资料来了解具体的内容。

在本节介绍的几种直线生成算法中，DDA 算法最简单，但是因为有多次浮点数乘法和除法运算，以及浮点数圆整运算，效率比较低。Bresenham 算法中的整数乘法计算都可以用移位代替，主要运算都采用了整数加法和减法运算，因此效率比较高，各种各样变形的 Bresenham 算法在计算机图形软件中得到了广泛的应用。理论上讲，两步算法以及四步算法效率应该更高一些，但是这两种算法需要做比较多的准备工作，且多是乘法和除法运算，因此在生成比较短的直线时，效率反而不如 Bresenham 算法。

14.3 圆生成算法

在平面解析几何中，圆的方程可以描述为$(x - x_0)^2 + (y - y_0)^2 = R^2$，其中$(x_0, y_0)$是圆心坐标，$R$ 是圆的半径，特别的，当(x_0, y_0)就是坐标中心点时，圆方程可以简化为 $x^2 + y^2 = R^2$。在计算机

图形学中，圆和直线一样，也存在点阵输出设备上显示或输出的问题，因此也需要一套光栅扫描转换算法。为了简化，我们先考虑圆心在原点的圆的生成，对于中心不是原点的圆，可以通过坐标的平移变换获得相应位置的圆。

14.3.1　圆的八分对称性

在进行扫描转换之前，需要了解一个圆的特性，就是圆的八分对成性。如图 14-9 所示，圆心位于原点的圆有四条对称轴 $x = 0$、$y = 0$、$x = y$ 和 $x = -y$，若已知圆弧上一点 $P(x, y)$，就可以得到其关于四条对称轴的七个对称点：$(x, -y)$、$(-x, y)$、$(-x, -y)$、(y, x)、$(y, -x)$、$(-y, x)$、$(-y, -x)$，这种性质称为八分对称性。因此只要能画出八分之一的圆弧，就可以利用对称性的原理得到整个圆。

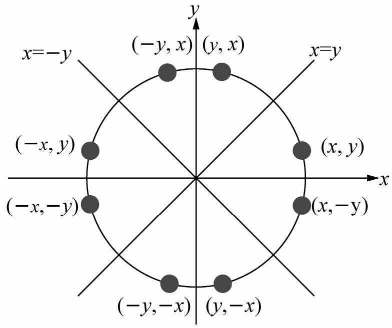


图 14-9　圆的八分对称性

有几种较容易的方法可以得到圆的扫描转换，首先介绍一下直角坐标法。已知圆方程：$x_2 + y_2 = R_2$，若取 x 作为自变量，解出 y，得到：

$$y = \pm\sqrt{R^2 - x^2}$$

在生成圆时先扫描转换四分之一的圆周，让自变量 x 从 0 到 R 以单位步长增加，在每一步时可解出 y，然后调用画点函数即可逐点画出圆。但这样做，由于有乘方和平方根运算，并且都是浮点运算，算法效率不高。而且当 x 接近 R 值时（圆心在原点），在圆周上的点$(R, 0)$附近，由于圆的斜率趋于无穷大，因浮点数取整需要四舍五入的缘故，使得圆周上有较大的间隙。接下来介绍一下极坐标法，假设直角坐标系上圆弧上一点 $P(x, y)$与 x 轴的夹角是 θ，则圆的极坐标方程为：

$$x = R\cos\theta$$

$$y = R\sin\theta$$

生成圆是利用圆的八分对称性，使自变量 θ 的取值范围为$(0, 45°)$就可以画出整圆。这个方法涉及三角函数计算和乘法运算，计算量较大。直角坐标法和极坐标法都是效率不高的算法，因此只是作为理论方法存在，在计算机图形学中基本不使用这两种方法生成圆。下面就介绍几种在计算机图形学中比较实用的圆的生成算法。

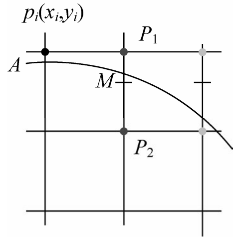


图 14-10　中点划线法示例

14.3.2 中点画圆法

首先是中点画圆法，考虑圆心在原点，半径为 R 的圆在第一象限内的八分之一圆弧，从点 $(0, R)$到点$(R/\sqrt{2}, R/\sqrt{2})$顺时针方向确定这段圆弧。假定某点 $P_i(x_i, y_i)$已经是该圆弧上最接近实际圆弧的点，那么 Pi 的下一个点只可能是正右方的 P_1 或右下方的 P_2 两者之一，如图 14-10 所示。

构造判别函数：

$$F(x, y) = x^2 + y^2 - R^2$$

当 $F(x, y)= 0$ 时，表示点在圆上，当 $F(x, y)> 0$，表示点在圆外，当 $F(x, y)< 0$ 时，表示点在圆内。如果 M 是 P_1 和 P_2 的中点，则 M 的坐标是$(x_i + 1, y_i - 0.5)$，当 $F(x_i + 1, y_i - 0.5)< 0$ 时，M 点在圆内，说明 P_1 点离实际圆弧更近，应该取 P_1 作为圆的下一个点。同理分析，当 $F(x_i + 1, y_i - 0.5)> 0$ 时，P_2 离实际圆弧更近，应取 P_2 作为下一个点。当 $F(x_i + 1, y_i - 0.5)= 0$ 时，P_1 和 P_2 都可以作为圆的下一个点，算法约定取 P_2 作为下一个点。

现在将 M 点坐标$(x_i + 1, y_i - 0.5)$带入判别函数 $F(x, y)$，得到判别式 d：

$$d = F(x_i + 1, y_i - 0.5)= (x_i + 1)^2 + (y_i - 0.5)^2 - R^2$$

若 $d < 0$，则取 P_1 为下一个点，此时 P_1 的下一个点的判别式为：

$$d' = F(x_i + 2, y_i - 0.5)= (x_i + 2)^2 + (y_i - 0.5)^2 - R^2$$

展开后将 d 带入可得到判别式的递推关系：

$$d' = d + 2x_i + 3$$

若 $d > 0$，则取 P_2 为下一个点，此时 P_2 的下一个点的判别式为：

$$d' = F(x_i + 2, y_i - 1.5)= (x_i + 2)^2 + (y_i - 1.5)^2 - R^2$$

展开后将 d 带入可得到判别式的递推关系：

$$d' = d + 2(x_i - y_i) + 5$$

特别的，在第一个象限的第一个点$(0, R)$时，可以推倒出判别式 d 的初始值 d_0：

$$d_0 = F(1, R - 0.5) = 1 - (R - 0.5)^2 - R^2 = 1.25 - R$$

根据上面的分析，可以写出中点画圆法的算法。考虑到圆心不在原点的情况，需要对计算出来的坐标进行了平移，下面就是通用的中点画圆法的源代码：

```
void MP_Circle(int xc , int yc , int r)
{
    int x, y;
    double d;

    x = 0;
    y = r;
```

```
    d = 1.25 - r;
    CirclePlot(xc , yc , x , y);
    while(x < y)
    {
        if(d < 0)
        {
            d = d + 2 * x + 3;
        }
        else
        {
            d = d + 2 * ( x - y ) + 5;
            y--;
        }
        x++;
        CirclePlot(xc , yc , x , y);
    }
}
```

参数 xc 和 yc 是圆心坐标，r 是半径，CirclePlot()函数是参照圆的八分对称性完成八个点的位置计算的辅助函数。

14.3.3 改进的中点画圆法——Bresenham 算法

中点画圆法中，计算判别式 d 使用了浮点运算，影响了圆的生成效率。如果能将判别式规约到整数运算，则可以简化计算，提高效率。于是人们针对中点画圆法进行了多种改进，其中一种方式是将 d 的初始值由 $1.25 - R$ 改成 $1 - R$，考虑到圆的半径 R 总是大于 2，因此这个修改不会影响 d 的初始值的符号，同时可以避免浮点运算。还有一种方法是将 d 的计算放大两倍，同时将初始值改成 $3 - 2R$，这样避免了浮点运算，乘二运算也可以用移位快速代替，采用 $3 - 2R$ 为初始值的改进算法，又称为 Bresenham 算法：

```
void Bresenham_Circle(int xc , int yc , int r)
{
    int x, y, d;

    x = 0;
    y = r;
    d = 3 - 2 * r;
    CirclePlot(xc , yc , x , y);
    while(x < y)
    {
        if(d < 0)
        {
            d = d + 4 * x + 6;
        }
        else
        {
            d = d + 4 * ( x - y ) + 10;
            y--;
        }
        x++;
        CirclePlot(xc , yc , x , y);
```

```
    }
}
```

14.3.4　正负判定画圆法

除了中点画圆算法，还有一种画圆算法也是利用当前点产生的圆函数进行符号判别，利用负反馈调整以决定下一个点的产生来直接生成圆弧，就是正负法，下面就介绍一下正负法的算法实现。

正负法根据圆函数：$F(x, y)=x^2+y^2-R^2$的值，将平面区域分成圆内和圆外，如图 14-11 所示。假设圆弧的生成方向是从 A 到 B 方向，当某个点 P_i 被确定以后，P_i 的下一个点 P_{i+1} 的取值就根据 $F(x_i, y_i)$的值进行判定，判定的原则如下。

- 当 $F(x_i, y_i)\leqslant 0$ 时：取 $x_{i+1}=x_i+1$，$y_{i+1}=y_i$。即向右走一步，从圆内走向圆外。对应图 14-11a 中的从 P_i 到 P_{i+1}。
- 当 $F(x_i, y_i)>0$ 时：取 $x_{i+1}=x_i$，$y_{i+1}=y_i-1$。即向下走一步，从圆外走向圆内。对应图 14-11b 中的从 P_i 到 P_{i+1}。

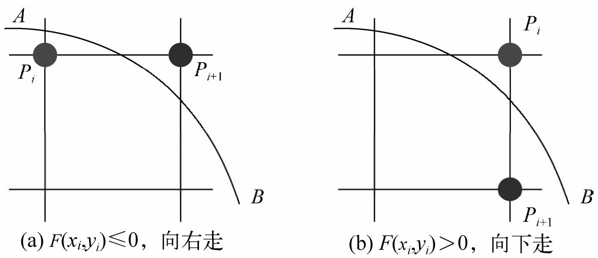


图 14-11　正负法判定示意图

由于下一个点的取向到底是向圆内走还是向圆外走取决于 $F(x_i, y_i)$的正负，因此称为正负法。对于判别式 $F(x_i, y_i)$的递推公式，也要分以下两种情况分别推算。

- 当 $F(x_i, y_i)\leqslant 0$ 时，P_i 的下一个点 P_{i+1} 取 $x_{i+1}=x_i+1$，$y_{i+1}=y_i$，判别式 $F(x_{i+1}, y_{i+1})$的推算过程是：

$$F(x_{i+1}, y_{i+1})= F(x_i+1,y_i) = (x_i+1)^2+y_i^2-R^2 = (x_i^2+y_i^2-R^2)+2x_i+1 = F(x_i, y_i)+2x_i+1$$

- 当 $F(x_i, y_i)>0$ 时，P_i 的下一个点 P_{i+1} 取 $x_{i+1}=x_i$，$y_{i+1}=y_i-1$，判别式 $F(x_{i+1}, y_{i+1})$ 的推算过程是：

$$F(x_{i+1}, y_{i+1})= F(x_i, y_i-1) = x_i^2+(y_i-1)^2-R^2 = (x_i^2+y_i^2-R^2)-2y_i+1 = F(x_i, y_i)-2y_i+1$$

设画圆的初始点是(0, R)，判定式的初始值是 0，正负法生成圆的算法如下：

```
void Pnar_Circle(int xc, int yc, int r)
{
    int x, y, f;
```

```
    x = 0;
    y = r;
    f = 0;
    while(x <= y)
    {
        CirclePlot(xc, yc, x, y);
        if(f <= 0)
        {
            f = f + 2 * x + 1;
            x++;
        }
        else
        {
            f = f - 2 * y + 1;
            y--;
        }
    }
}
```

改进的中点划线算法和正负法虽然都避免了浮点运算，并且计算判别式时用到的乘法都是乘 2 运算，可以用移位代替，但是实际效率却有很大差别。因为正负法并不是严格按照 x 方向步进的，因此就会出现在某个点的下一个点在两个位置上重复画点的问题，增加了不必要的计算。此外，从生成圆的质量看，中点画圆法和改进的中点画圆法都比正负法效果好。

14.4 椭圆生成算法

椭圆和直线、圆一样，是图形学领域中的一种常见图元，椭圆的生成算法（光栅转换算法）也是图形学软件中最常见的生成算法之一。在平面解析几何中，椭圆的方程可以描述为：

$$\frac{(x-x_0)^2}{a^2}+\frac{(y-y_0)^2}{b^2}=1$$

其中(x_0, y_0)是圆心坐标，a 和 b 是椭圆的长短轴，特别的，当(x_0, y_0)就是坐标中心点时，椭圆方程可以简化为：

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$$

在计算机图形学中，椭圆图形也存在在点阵输出设备上显示或输出的问题，因此也需要一套光栅扫描转换算法。为了简化，我们先考虑圆心在原点的椭圆的生成，对于中心不是原点的椭圆，可以通过坐标的平移变换获得相应位置的椭圆。

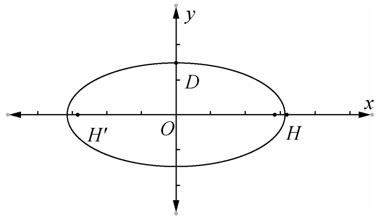


图 14-12 椭圆的对称性

在进行扫描转换之前，需要了解一下椭圆的对称性，

如图 14-12 所示。中心在原点。焦点在坐标轴上的标准椭圆具有 X 轴对称、Y 轴对称和原点对称特性，已知椭圆上第一象限的 P 点坐标是(x, y)，则椭圆在另外三个象限的对称点分别是$(x, -y)$、$(-x, y)$和$(-x, -y)$。因此，只要画出第一象限的四分之一椭圆，就可以利用这三个对称性得到整个椭圆。

在光栅设备上输出椭圆有很多种方法，可以根据直角平面坐标方程直接求解点坐标，也可以利用极坐标方程求解，但是因为涉及浮点数取整，效果都不好，一般都不使用直接求解的方式。本文就介绍几种计算机图形学中两种比较常用的椭圆生成方法：中点画椭圆算法和 Bresenham 椭圆生成算法。

14.4.1 中点画椭圆法

中点在坐标原点，焦点在坐标轴上（轴对齐）的椭圆的平面方程也可以转化为如下非参数化方程形式：

$$F(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = 0 \tag{14-18}$$

无论是中点画线算法、中点画圆算法还是本节要介绍的中点画椭圆算法，对选择 x 方向像素 Δ 增量还是 y 方向像素 Δ 增量都是很敏感的。举个例子，如果某段圆弧上，x 方向上增量+1 个像素时，y 方向上的增量如果 < 1，则比较适合用中点算法，如果 y 方向上的增量 > 1，就会产生一些跳跃的点,最后生成的光栅位图圆弧会有一些突变的点，看起来好像不在圆弧上。因此，对于中点画圆弧算法，要区分出椭圆弧上哪段 Δx 增量变化显著，哪段 Δy 增量变化显著，然后区别对待。由于椭圆的对称性，我们只考虑第一象限的椭圆圆弧，如图 14-13 所示。

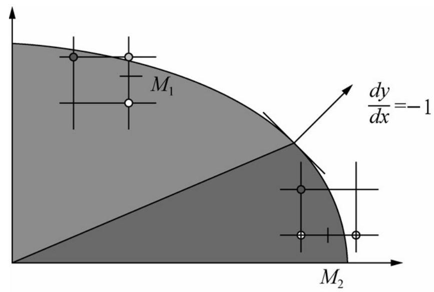


图 14-13 第一象限椭圆弧示意图

定义椭圆弧上某点的切线法向量 N 如下：

$$N(x, y) = \frac{\partial F(x, y)}{\partial x} i + \frac{\partial F(x, y)}{\partial y} j = 2b^2 x_i + 2a^2 y_j$$

对方程(14-18)分别对 x、y 求偏导，最后得到椭圆弧上(x,y)点处的法向量是（$2b^2x, 2a^2y$）。$dy/dx = -1$ 的点是椭圆弧上的分界点。此点之上的部分（浅色部分）椭圆弧法向量的 y 分量比较大，即 $2b^2(x + 1) < 2a^2(y - 0.5)$；此点之下的部分（深色部分）椭圆弧法向量的 x 分量比较大，即 $2b^2(x + 1) > 2a^2(y - 0.5)$。

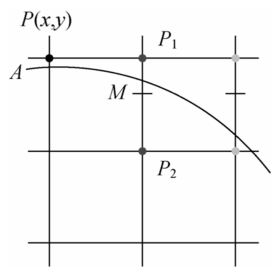


图 14-14 中点法对上部区域处理示意图

对于图 14-13 中浅色标识的上部区域，y 方向每变化

1 个单位，x 方向变化大于一个单位，因此中点算法需要沿着 x 方向步进画点，x 每次增量加 1，求 y 的值。同理，对于图 14-13 中深色标识的下部区域，中点算法沿着 y 方向反向步进，y 每次减 1，求 x 的值。先来讨论上部区域椭圆弧的生成，如图 14-14 所示。假设当前位置是 $P(x_i, y_i)$，则下一个可能的点就是 P 点右边的 $P_1(x_i+1, y_i)$点或右下方的 $P_2(x_i+1, y_i-1)$点，取舍的方法取决于判别式 d_i，d_i的定义如下：

$$d_i = F(x_i+1, y_i - 0.5) = b^2(x_i+1)^2 + a^2(y_i - 0.5)^2 - a^2b^2$$

若 $d_i < 0$，表示像素点 P_1 和 P_2 的中点在椭圆内，这时可取 P_1 为下一个像素点。此时 $x_{i+1} = x_i + 1$，$y_{i+1} = y_i$，代入判别式 d_i 得到 d_{i+1}：

$$d_{i+1} = F(x_{i+1}+1, y_{i+1} - 0.5) = b^2(x_i+2)^2 + a^2(y_i - 0.5)^2 - a^2b^2 = d_i + b^2(2x_i + 3)$$

计算出 d_i 的增量是 $b^2(2x_i + 3)$。同理，若 $d_i \geqslant 0$，表示像素点 P_1 和 P_2 的中点在椭圆外，这时应当取 P_2 为下一个像素点。此时 $x_{i+1} = x_i + 1$，$y_{i+1} = y_i - 1$，代入判别式 d_i 得到 d_{i+1}：

$$d_{i+1} = F(x_{i+1}+1, y_{i+1} - 0.5) = b^2(x_i+2)^2 + a^2(y_i -1.5)^2 - a^2b^2 = d_1 + b^2(2x_i+3) + a^2(-2y_i+2)$$

计算出 d_i 的增量是 $b^2(2x_i+3)+a^2(-2y_i+2)$。计算 d_i 的增量的目的是减少计算量，提高算法效率，每次判断一个点时，不必完整的计算判别式 d_i，只需在上一次计算出的判别式上增加一个增量即可。

接下来看看下部区域椭圆弧的生成，如图 14-15 所示。假设当前位置是 $P(x_i, y_i)$，则下一个可能的点就是 P 点左下方的 $P_1(x_i-1, y_i-1)$点或下方的 $P_2(x_i, y_i-1)$点，取舍的方法同样取决于判别式 d_i，d_i的定义如下：

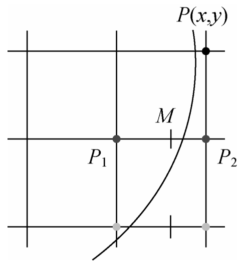


图 14-15　中点法对下部区域处理示意图

$$d_i = F(x_i+0.5, y_i-1) = b^2(x_i+0.5)^2 + a^2(y_i-1)^2 - a^2b^2$$

若 $d_i < 0$，表示像素点 P_1 和 P_2 的中点在椭圆内，这时可取 P_2 为下一个像素点。此时 $x_{i+1} = x_i + 1$，$y_{i+1} = y_i - 1$，代入判别式 d_i 得到 d_{i+1}：

$$d_{i+1} = F(x_{i+1}+0.5, y_{i+1}-1) = b^2(x_i+1.5)^2 + a^2(y_i-2)^2 - a^2b^2 = d_i + b^2(2x_i+2)+a^2(-2y_i+3)$$

计算出 d_i 的增量是 $b^2(2x_i+2)+a^2(-2y_i+3)$。同理，若 $d_i \geqslant 0$，表示像素点 P_1 和 P_2 的中点在椭圆外，这时应当取 P_1 为下一个像素点。此时 $x_{i+1} = x_i$，$y_{i+1} = y_i - 1$，代入判别式 d_i 得到 d_{i+1}：

$$d_{i+1} = F(x_{i+1}+0.5, y_{i+1}-1) = b^2(x_i+0.5)^2 + a^2(y_i-2)^2 - a^2b^2 = d_1 + a^2(-2y_i+3)$$

计算出 d_i 的增量是 $a^2(-2y_i+3)$。

中点画椭圆算法从$(0, b)$点开始，第一个中点是$(1, b - 0.5)$，判别式 d 的初始值是：

$$d_0 = F(1, b - 0.5) = b^2 + a^2(-b+0.25)$$

上部区域生成算法的循环终止条件是：$2b^2(x + 1) \geqslant 2a^2(y - 0.5)$，下部区域的循环终止条件是 $y =$

0，至此，中点画椭圆算法就可以完整给出了：

```
void MP_Ellipse(int xc , int yc , int a, int b)
{
    double sqa = a * a;
    double sqb = b * b;

    double d = sqb + sqa * (-b + 0.25);
    int x = 0;
    int y = b;
    EllipsePlot(xc, yc, x, y);
    while( sqb * (x + 1) < sqa * (y - 0.5))
    {
        if (d < 0)
        {
            d += sqb * (2 * x + 3);
        }
        else
        {
            d += (sqb * (2 * x + 3) + sqa * (-2 * y + 2));
            y--;
        }
        x++;
        EllipsePlot(xc, yc, x, y);
    }
    d = (b * (x + 0.5)) * 2 + (a * (y - 1)) * 2 - (a * b) * 2;
    while(y > 0)
    {
        if (d < 0)
        {
            d += sqb * (2 * x + 2) + sqa * (-2 * y + 3);
            x++;
        }
        else
        {
            d += sqa * (-2 * y + 3);
        }
        y--;
        EllipsePlot(xc, yc, x, y);
    }
}
```

EllipsePlot()函数利用椭圆的三个对称性，一次完成四个对称点的绘制，因为简单，此处就不再列出代码。

14.4.2 Bresenham 椭圆算法

中点画椭圆法中，计算判别式 d 使用了浮点运算，影响了椭圆的生成效率。如果能将判别式规约到整数运算，则可以简化计算，提高效率。于是人们针对中点画椭圆法进行了多种改进，提出了很多种中点生成椭圆的整数型算法，Bresenham 椭圆生成算法就是其中之一。

在生成椭圆上部区域时，以 x 轴为步进方向，如图 14-16a 所示，当 x 步进到 $x+1$ 时，需要判断 y 的值是保持不变还是步进到 $y–1$，Bresenham 算法定义判别式为：

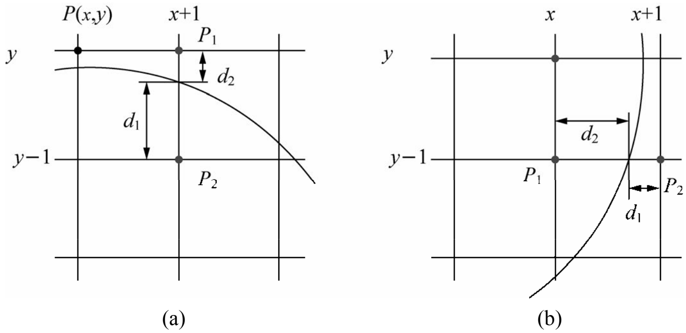


图 14-16 Bresenham 椭圆生成算法判别式

$$D = d_1 - d_2$$

如果 $D < 0$，则取 P_1 为下一个点，否则，取 P_2 为下一个点。采用判别式 D，避免了中点算法因 $y–0.5$ 而引入的浮点运算，使得判别式规约为全整数运算，算法效率得到了很大的提升。根据椭圆方程，可以计算出 d_1 和 d_2 分别是：

$$d_1 = a^2(y_i^2 - y^2)$$

$$d_2 = a^2(y^2 - y_{i+1}^2)$$

以$(0, b)$作为椭圆上部区域的起点，将其代入判别式 D 可以得到如下递推关系：

$$D_{i+1} = D_i + 2b^2(2x_i + 3)\ (D_i < 0)$$

$$D_{i+1} = D_i + 2b^2(2x_i + 3) - 4a^2(y_i - 1)\ (D_i \geqslant 0)$$

$$D_0 = 2b^2 - 2a^2b + a^2$$

在生成椭圆下部区域时，以 y 轴为步进方向，如图 14-16b 所示，当 y 步进到 $y–1$ 时，需要判断 x 的值是保持不变还是步进到 $x+1$，对于下部区域，计算出 d_1 和 d_2 分别是：

$$d_1 = b^2(x_{i+1}^2 - x^2)$$

$$d_2 = b^2(x^2 - x_i^2)$$

以(x_p, y_p)作为椭圆下部区域的起点，将其代入判别式 D 可以得到如下递推关系：

$$D_{i+1} = D_i - 4a^2(y_i - 1) + 2a^2 \qquad (D_i < 0)$$

$$D_{i+1} = D_i + 2b^2(x_i + 1) - 4a^2(y - 1) + 2a^2 + b^2 \qquad (D_i \geqslant 0)$$

$$D_0 = b^2(x_p + 1)^2 + b^2x_p^2 - 2a^2b^2 + 2a^2(y_p - 1)^2$$

根据以上分析，Bresenham 椭圆生成算法的实现就比较简单了：

```
void Bresenham_Ellipse(int xc , int yc , int a, int b)
{
    int sqa = a * a;
    int sqb = b * b;

    int x = 0;
    int y = b;
    int d = 2 * sqb - 2 * b * sqa + sqa;
    EllipsePlot(xc, yc, x, y);
    int P_x = ROUND_INT( (double)sqa/sqrt((double)(sqa+sqb)) );
    while(x <= P_x)
    {
        if(d < 0)
        {
            d += 2 * sqb * (2 * x + 3);
        }
        else
        {
            d += 2 * sqb * (2 * x + 3) - 4 * sqa * (y - 1);
            y--;
        }
        x++;
        EllipsePlot(xc, yc, x, y);
    }

    d = sqb * (x * x + x) + sqa * (y * y - y) - sqa * sqb;
    while(y >= 0)
    {
        EllipsePlot(xc, yc, x, y);
        y--;
        if(d < 0)
        {
            x++;
            d = d - 2 * sqa * y - sqa + 2 * sqb * x + 2 * sqb;
        }
        else
        {
            d = d - 2 * sqa * y - sqa;
        }
    }
}
```

14.5 多边形区域填充算法

平面区域填充算法是计算机图形学领域的一个很重要的算法，区域填充即给出一个区域的边界（也可以是没有边界，只是给出指定颜色），要求将边界范围内的所有像素单元都修改成指定的颜色（也可能是图案填充）。区域填充中最常用的是多边形填色，本节我们就讨论几种多边形区域填充算法。

14.5.1 种子填充算法

如果要填充的区域是以图像元数据方式给出的，通常使用*种子填充算法*进行区域填充。种子填充算法需要给出图像数据的区域，以及区域内的一个点，这种算法比较适合人机交互方式进行的图像填充操作，不适合计算机自动处理和判断填色。根据对图像区域边界定义方式以及对点的颜色修改方式，种子填充又可细分为几类，比如注入填充算法、边界填充算法以及为减少递归和压栈次数而改进的扫描线种子填充算法等。

所有种子填充算法的核心其实就是一个递归算法，都是从指定的种子点开始，向各个方向上搜索，逐个像素进行处理，直到遇到边界，各种种子填充算法只是在处理颜色和边界的方式上有所不同。在开始介绍种子填充算法之前，首先也介绍两个概念，就是“4-联通算法”和“8-联通算法”。既然是搜索就涉及搜索的方向问题，从区域内任意一点出发，如果只是通过上、下、左、右四个方向搜索到达区域内的任意像素，则用这种方法填充的区域就称为*四连通域*，这种填充方法就称为 *4-联通算法*。如果从区域内任意一点出发，通过上、下、左、右、左上、左下、右上和右下全部八个方向到达区域内的任意像素，则这种方法填充的区域就称为*八连通域*，这种填充方法就称为 *8-联通算法*。如图 14-17a 所示，假设中心的深灰色点是当前处理的点，如果是 4-联通算法，则只搜索处理周围深灰色标识的四个点；如果是 8-联通算法则除了处理上、下、左、右四个深灰色标识的点，还搜索处理四个浅灰色标识的点。假如都是从白色点开始填充，两种搜索算法的填充效果分别如图 14-17b 和图 14-17c 所示。

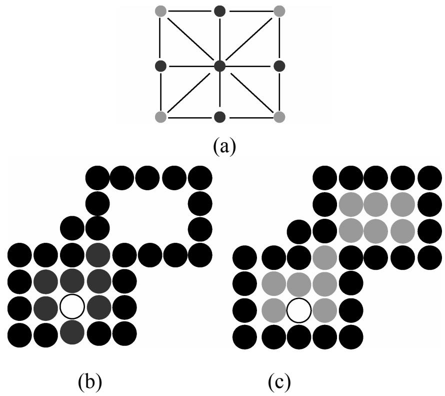


图 14-17 “4-联通”和“8-联通”填充效果图

1. 注入填充算法（Flood Fill Algorithm）

*注入填充算法*不特别强调区域的边界，它只是从指定位置开始，将所有联通区域内某种指定颜色的点都替换成另一种颜色，从而实现填充效果。注入填充算法能够实现颜色替换之类的功能，这在图像处理软件中都得到了广泛的应用。注入填充算法的实现非常简单，核心就是递归和搜索，以下就是注入填充算法的一个实现：

```
void FloodSeedFill(int x, int y, int old_color, int new_color)
{
    if(GetPixelColor(x, y) == old_color)
    {
        SetPixelColor(x, y, new_color);
        for(int i = 0; i < COUNT_OF(direction_8); i++)
        {
            FloodSeedFill(x + direction_8[i].x_offset,
                y + direction_8[i].y_offset, old_color, new_color);
        }
    }
```

```
}
```

for 循环实现了向 8 个联通方向的递归搜索，秘密就在于 direction_8 的定义：

```
typedef struct tagDIRECTION
{
    int x_offset;
    int y_offset;
}DIRECTION;

DIRECTION direction_8[] = { {-1, 0}, {-1, 1}, {0, 1}, {1, 1}, {1, 0}, {1, -1}, {0, -1}, {-1, -1} };
```

这个是搜索类算法中常用的技巧，在本书第 1 章已经提到过类似的方法。只要将其替换成如下 direction_4 的定义，就可以将算法改成 4 个联通方向填充算法：

```
DIRECTION direction_4[] = { {-1, 0}, {0, 1}, {1, 0}, {0, -1} };
```

2. 边界填充算法

*边界填充算法*与注入填充算法的本质其实是一样的，都是递归和搜索，区别只在于对边界的确认，也就是递归的结束条件不一样。注入填充算法没有边界的概念，只是对联通区域内指定的颜色进行替换，而边界填充算法恰恰强调边界的存在，只要是边界内的点无论是什么颜色，都替换成指定的颜色。边界填充算法的应用也非常广泛，画图软件中的“油漆桶”功能就是边界填充算法的例子。以下就是边界填充算法的一个实现。

```
void BoundarySeedFill(int x, int y, int new_color, int boundary_color)
{
    int curColor = GetPixelColor(x, y);
    if( (curColor != boundary_color)
        && (curColor != new_color) )
    {
        SetPixelColor(x, y, new_color);
        for(int i = 0; i < COUNT_OF(direction_8); i++)
        {
            BoundarySeedFill(x + direction_8[i].x_offset,
                y + direction_8[i].y_offset, new_color, boundary_color);
        }
    }
}
```

3. 扫描线种子填充算法

前面介绍的两种种子填充算法的优点是非常简单，缺点是使用了递归算法，这不但需要大量栈空间来存储相邻的点，而且效率不高。为了减少算法中的递归调用，节省栈空间的使用，人们提出了很多改进算法，其中一种就是*扫描线种子填充算法*。该算法不再采用递归的方式处理 4-联通和 8-联通的相邻点，而是通过沿水平扫描线填充像素段，一段一段地来处理 4-联通和 8-联通的相邻点。这样算法处理过程中就只需要将每个水平像素段的起始点位置压入一个特殊的栈，而不需要象递归算法那样将当前位置周围尚未处理的所有相邻点都压入堆栈，从而可以节省堆栈空间。应该说，扫描线填充算法只是一种避免递归，提高效率的思想，前面提到的注入填充算法和边界填充算法都可以改进成扫描线填充算法，下面介绍的就是结合了边界填充算法的扫描线种子填充算法。

扫描线种子填充算法的基本过程如下：当给定种子点(x, y)时，首先分别向左和向右两个方向填充种子点所在扫描线上的位于给定区域的一个区段，同时记下这个区段的范围[xLeft, xRight]，然后确定与这一区段相连通的上、下两条扫描线上位于给定区域内的区段，并依次保存下来。反复这个过程，直到填充结束。扫描线种子填充算法可由下列四个步骤实现。

(1) 初始化一个空的栈用于存放种子点，将种子点(x, y)入栈；

(2) 判断栈是否为空，如果栈为空则结束算法，否则取出栈顶元素作为当前扫描线的种子点(x, y)，y 是当前的扫描线；

(3) 从种子点(x, y)出发，沿当前扫描线向左、右两个方向填充，直到边界。分别标记区段的左、右端点坐标为 xLeft 和 xRight；

(4) 分别检查与当前扫描线相邻的 $y - 1$ 和 $y + 1$ 两条扫描线在区间[xLeft, xRight]中的像素，从 xLeft 开始向 xRight 方向搜索，若存在非边界且未填充的像素点，则找出这些相邻的像素点中最右边的一个，并将其作为种子点压入栈中，然后返回第(2)步；

这个算法中最关键的是第(4)步，就是从当前扫描线的上一条扫描线和下一条扫描线中寻找新的种子点。这里比较难理解的一点就是为什么只是检查新扫描线上区间[xLeft, xRight]中的像素？如果新扫描线的实际范围比这个区间大（而且不连续）怎么处理？我查了很多计算机图形学的书和论文，好像都没有对此做过特殊说明，这使得很多人在学习这门课程时对此有挥之不去的疑惑。本着"毁人"不倦的思想，我们就啰嗦解释一下，希望能解除大家的疑惑。

如果新扫描线上实际点的区间比当前扫描线的[xLeft, xRight]区间大，而且是连续的情况下，算法的第(3)步就处理了这种情况。如图 14-18 所示，假设当前处理的扫描线是白色点所在的第 7 行，则经过第(3)步处理后可以得到一个区间[6,10]。然后第 4 步操作，从相邻的第 6 行和第 8 行两条扫描线的第 6 列开始向右搜索，确定浅灰色的两个点分别是第 6 行和第 8 行的种子点，于是按照顺序将(6, 10)和(8, 10)两个种子点入栈。接下来的循环会处理(8, 10)这个种子点，根据算法第(3)步说明，会从(8, 10)开始向左和向右填充，由于中间没有边界点，因此填充会直到遇到边界为止，所以尽管第 8 行实际区域比第 7 行的区间[6,10]大，但是仍然得到了正确的填充。

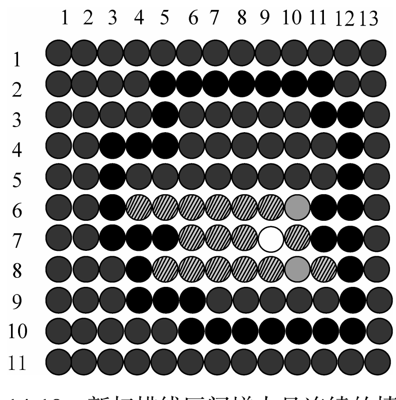


图 14-18 新扫描线区间增大且连续的情况

如果新扫描线上实际点的区间比当前扫描线的[*x*Left, *x*Right]区间大，而且中间有边界点的情况，算法又是怎么处理呢？算法描述中虽然没有明确对这种情况的处理方法，但是第(4)步确定上、下相邻扫描线的种子点的方法，以及靠右取点的原则，实际上暗含了从相邻扫描线绕过障碍点的方法。下面以图 14-19 为例说明，算法第(3)步处理完第 5 行后，确定了区间[7, 9]，相邻的第 4 行虽然实际范围比区间[7, 9]大，但是因为被(4, 6)这个边界点阻碍，使得在确定种子点(4, 9)后向左填充只能填充右边的第 7 列到第 10 列之间的区域，而左边的第 3 列到第 5 列之间的区域没有填充。虽然作为第 5 行的相邻行，第一次对第 4 行的扫描根据靠右原则只确定了(4, 9)一个种子点。但是对第 3 行处理完后，第 4 行的左边部分作为第 3 行下边的相邻行，再次得到扫描的机会。第 3 行的区间是[3, 9]，向左跨过了第 6 列这个障碍点，第 2 次扫描第 4 行的时候就从第 3 列开始，向右找，可以确定种子点(4, 5)。这样第 4 行就有了两个种子点，就可以被完整地填充了。

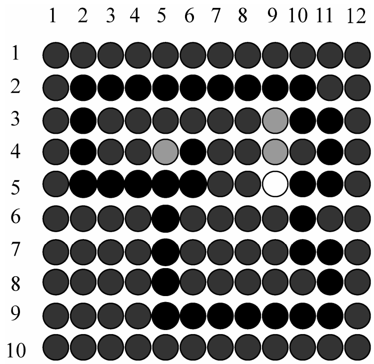


图 14-19　新扫描线区间增大且不连续的情况

由此可见，对于有障碍点的行，通过相邻边的关系，可以跨越障碍点，通过多次扫描得到完整的填充，算法已经隐含了对这种情况的处理。根据本节总结的四个步骤，扫描线种子填充算法的实现如下：

```
void ScanLineSeedFill(int x, int y, int new_color, int boundary_color)
{
    std::stack<POINT> stk;

    stk.push(POINT(x, y)); //第 1 步，种子点入站
    while(!stk.empty())
    {
        POINT seed = stk.top(); //第 2 步，取当前种子点
        stk.pop();

        //第 3 步，向左右填充
        int count = FillLineRight(seed.x, seed.y, new_color, boundary_color);//向右填充
        int xRight = seed.x + count - 1;
        count = FillLineLeft(seed.x - 1, seed.y, new_color, boundary_color);//向左填充
        int xLeft = seed.x - count;

        //第 4 步，处理相邻两条扫描线
```

```
        SearchLineNewSeed(stk, xLeft, xRight, seed.y - 1, new_color, boundary_color);
        SearchLineNewSeed(stk, xLeft, xRight, seed.y + 1, new_color, boundary_color);
    }
}
```

FillLineRight()和 FillLineLeft()两个函数就是从种子点分别向右和向左填充颜色，直到遇到边界点，同时返回填充的点的个数。这两个函数返回填充点的个数是为了正确调整当前种子点所在的扫描线的区间[*x*Left, *x*Right]。SearchLineNewSeed()函数完成算法第 (4) 步所描述的操作，就是在新扫描线上寻找种子点，并将种子点入栈，新扫描线的区间是 xLeft 和 xRight 参数确定的：

```
void SearchLineNewSeed(std::stack<POINT>& stk, int xLeft, int xRight,
    int y, int new_color, int boundary_color)
{
    int xt = xLeft;
    bool findNewSeed = false;

    while(xt <= xRight)
    {
        findNewSeed = false;
        while(IsPixelValid(xt, y, new_color, boundary_color) && (xt < xRight))
        {
            findNewSeed = true;
            xt++;
        }
        if(findNewSeed)
        {
            if(IsPixelValid(xt, y, new_color, boundary_color) && (xt == xRight))
                stk.push(POINT(xt, y));
            else
                stk.push(POINT(xt - 1, y));
        }

        /*向右跳过内部的无效点（处理区间右端有障碍点的情况）*/
        int xspan = SkipInvalidInLine(xt, y, xRight, new_color, boundary_color);
        xt += (xspan == 0) ? 1 : xspan;
        /*处理特殊情况,以退出 while(x<=xright)循环*/
    }
}
```

最外层的 while 循环是为了保证区间[*x*Left, *x*Right]右端被障碍点分隔成多段的情况能够得到正确处理，通过外层 while 循环，可以确保为每一段都找到一个种子点（对于障碍点在区间左端的情况，请参考图 14-19 所示实例的解释，是隐含在算法中完成的）。内层的 while 循环只是为了找到每一段最右端的一个可填充点作为种子点。SkipInvalidInLine()函数的作用就是跳过区间内的障碍点，确定下一个分隔段的开始位置。循环内的最后一行代码有点奇怪，其实只是用了一个小“诡计”，确保在遇到真正的边界点时循环能够正确退出。这不是一个值得称道的做法，实现此类软件控制有更好的方法，这里这样做的目的只是为了使代码简短一些，让读者把注意力集中在算法处理逻辑上，而不是冗杂难懂的循环控制条件上。

算法的实现其实就在 ScanLineSeedFill()和 SearchLineNewSeed()两个函数中，神秘的扫描线种

子填充算法也并不复杂，对吧？至此，种子填充算法的几种常见算法都已经介绍完毕，接下来将介绍两种适合矢量图形区域填充的填充算法，分别是扫描线算法和边标志填充算法，注意适合矢量图形的扫描线填充算法有时又称为“有序边表法”，和扫描线种子填充算法是有区别的。

14.5.2 扫描线填充算法

*扫描线填充算法*适合对矢量图形进行区域填充，只需要知道多边形区域的几何位置，不需要指定种子点，适合计算机自动进行图形处理的场合使用，比如电脑游戏和三维 CAD 软件的渲染等。

对矢量多边形区域填充，算法核心还是求交。14.1.6 节给出了判断点与多边形关系的算法——扫描交点的奇偶数判断算法，利用此算法可以判断一个点是否在多边形内，也就是是否需要填充，但是实际工程中使用的填充算法都是只使用求交的思想，并不直接使用这种求交算法。究其原因，除了算法效率问题之外，还存在一个光栅图形设备和矢量之间的转换问题。比如某个点位于非常靠近边界的临界位置，用矢量算法判断这个点应该是在多边形内，但是光栅化后，这个点在光栅图形设备上看就有可能是在多边形外边（矢量点没有大小概念，光栅图形设备的点有大小概念），因此，适用于矢量图形的填充算法必须适应光栅图形设备。

1. 扫描线填充算法的基本思想

扫描线填充算法的基本思想是：用水平扫描线从上到下（或从下到上）扫描由多条首尾相连的线段构成的多边形，每根扫描线与多边形的某些边产生一系列交点。将这些交点按照 x 坐标排序，将排序后的点两两成对，作为线段的两个端点，以所填的颜色画水平直线。多边形被扫描完毕后，颜色填充也就完成了。扫描线填充算法也可以归纳为以下 4 个步骤。

(1) 求交，计算扫描线与多边形的交点。

(2) 交点排序，对第(2)步得到的交点按照 x 值从小到大进行排序。

(3) 颜色填充，对排序后的交点两两组成一个水平线段，以画线段的方式进行颜色填充。

(4) 是否完成多边形扫描？如果是就结束算法，如果不是就改变扫描线，然后转第(1)步继续处理。

整个算法的关键是第(1)步，需要用尽量少的计算量求出交点，还要考虑交点是线段端点的特殊情况，最后，交点的步进计算最好是整数，便于光栅设备输出显示。

对于每一条扫描线，如果每次都按照正常的线段求交算法进行计算，则计算量大，而且效率低下，如图 14-20 所示，观察多边形与扫描线的交点情况，可以得到以下两个特点。

- 每次只有相关的几条边可能与扫描线有交点，不必对所有的边进行求交计算；
- 相邻的扫描线与同一直线段的交点存在步进关系，这个关系与直线段所在直线的斜率有关。

第一个特点是显而易见的，为了减少计算量，扫描线算法需要维护一张由“活动边”组成的表，称为*活动边表*（AET）。例如扫描线 4 的活动边表由 P_1P_2 和 P_3P_4 两条边组成，而扫描线 7 的活动边表由 P_1P_2、P_6P_1、P_5P_6 和 P_4P_5 四条边组成。

第二个特点可以进一步证明，假设当前扫描线与多边形的某一条边的交点已经通过直线段求交算法计算出来，得到交点的坐标为(x, y)，则下一条扫描线与这条边的交点不需要再求交计算，通过步进关系可以直接得到新交点坐标为$(x + \Delta x, y + 1)$。前面提到过，步进关系Δx是个常量，与直线的斜率有关，下面就来推导这个Δx。

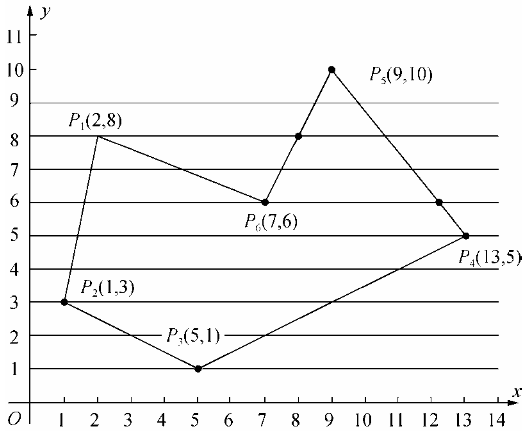


图 14-20　多边形与扫描线示意图

假设多边形某条边所在的直线方程是：$ax + by + c = 0$，扫描线 y_i 和下一条扫描线 y_{i+1} 与该边的两个交点分别是(x_i, y_i)和(x_{i+1}, y_{i+1})，则可得到以下两个等式：

$$ax_i + by_i + c = 0 \tag{14-19}$$

$$ax_{i+1} + by_{i+1} + c = 0 \tag{14-20}$$

式(14-19)经过变换可以得到式(14-21)：

$$x_i = -(by_i + c) / a \tag{14-21}$$

同样，式(14-20)经过变换可以得到式(14-22)：

$$x_{i+1} = -(by_{i+1} + c) / a \tag{14-22}$$

由式(14-22)与式(14-21)进行等式相减，得到式(14-23)：

$$x_{i+1} - x_i = -b\,(y_{i+1} - y_i) / a \tag{14-23}$$

由于扫描线存在 $y_{i+1} = y_i + 1$ 的关系，将其代入式(14-23)后可得式(14-24)：

$$x_{i+1} - x_i = -b / a \tag{14-24}$$

即$\Delta x = -b / a$，是个常量（直线斜率的倒数）。

活动边表是扫描线填充算法的核心，整个算法都是围绕着这张表进行处理的。要完整地定义活动边表，需要先定义边的数据结构。每条边都和扫描线有个交点，扫描线填充算法只关注交点

的 x 坐标。每当处理下一条扫描线时，根据Δx 直接计算出新扫描线与边的交点 x 坐标，可以避免复杂的求交计算。一条边不会一直待在活动边表中，当扫描线与之没有交点时，要将其从活动边表中删除，判断是否有交点的依据就是看扫描线 y 是否大于这条边两个端点的 y 坐标值，为此，需要记录边的 y 坐标的最大值。根据以上分析，边的数据结构可以定义如下：

```
typedef struct tagEDGE
{
    double xi;
    double dx;
    int ymax;
}EDGE;
```

根据 EDGE 的定义，扫描线 4 和扫描线 7 的活动边表就如图 14-21 所示。

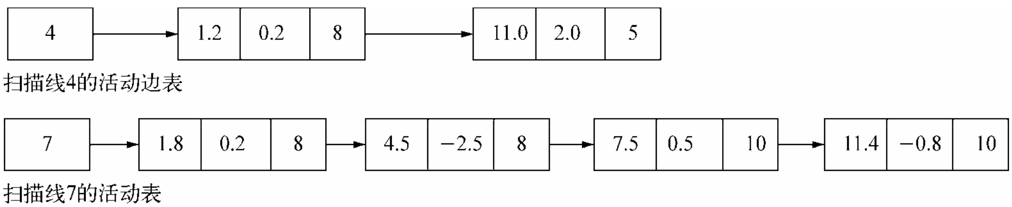


图 14-21 扫描线的活动边表示意图

前面提到过，扫描线算法的核心就是围绕活动边表展开的，为了方便活性边表的建立与更新，我们为每一条扫描线建立一个新边表（NET），存放该扫描线第一次出现的边。当算法处理到某条扫描线时，就将这条扫描线的新边表中的所有边逐一插入到活动边表中。新边表通常在算法开始时建立，建立新边表的规则就是：如果某条边的较低端点（y 坐标较小的那个点）的 y 坐标与扫描线 y 相等，则该边就是扫描线 y 的新边，应该加入扫描线 y 的新边表。上例中各扫描线的新边表如图 14-22 所示。

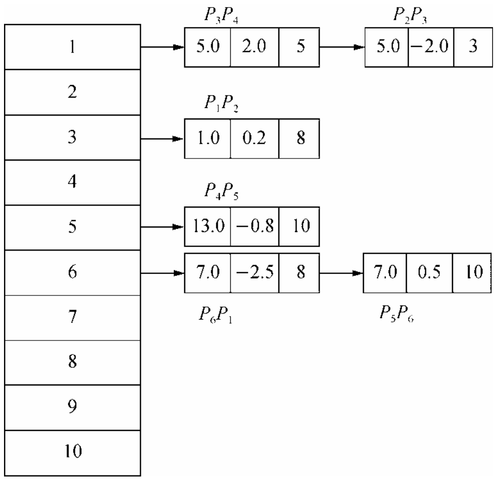


图 14-22 各扫描边的新边表

讨论完活动边表和新边表，就可以开始算法的具体实现了，但是在进一步详细介绍实现算法之前，还有以下三个关键的细节问题需要明确。

(1) *多边形顶点处理*。在对多边形的边进行求交的过程中，在两条边相连的顶点处会出现一些特殊情况，因为此时两条边会和扫描线各求的一个交点，也就是说，在顶点位置会出现两个交点。当出现这种情况的时候，会对填充产生影响，因为填充的过程是成对选择交点的过程，错误地计算交点个数，会造成填充异常。

假设多边形按照顶点 P_1、P_2 和 P_3 的顺序产生两条相邻的边，P_2 就是所说的顶点。多边形的顶点一般有四种情况，如图 14-23 所展示的那样，分别被称为左顶点、右顶点、上顶点和下顶点，它们的坐标满足以下关系。

左顶点——P_1、P_2 和 P_3 的 y 坐标满足条件：$y_1 < y_2 < y_3$；

右顶点——P_1、P_2 和 P_3 的 y 坐标满足条件：$y_1 > y_2 > y_3$；

上顶点——P_1、P_2 和 P_3 的 y 坐标满足条件：$y_2 > y_1$ && $y_2 > y_3$；

下顶点——P_1、P_2 和 P_3 的 y 坐标满足条件：$y_2 < y_1$ && $y_2 < y_3$；

对于左顶点和右顶点的情况，如果不做特殊处理会导致奇偶奇数错误，常采用的修正方法是修改以顶点为终点的那条边的区间，将顶点排除在区间之外，也就是删除这条边的终点，这样在计算交点时，就可以少计算一个交点，平衡和交点奇偶个数。结合前面定义的“边”数据结构：EDGE，只要将该边的 ymax 修改为 ymax−1 就可以了。

对于上顶点和下顶点，一种处理方法是将交点计算为 0 个，也就是修正两条边的区间，将交点从两条边中排除；另一种处理方法是不做特殊处理，就计算 2 个交点，这样也能保证交点奇偶个数平衡。

(2) *水平边的处理*。水平边与扫描线重合，会产生很多交点，通常的做法是将水平边直接画出（填充），然后在后面的处理中忽略水平边，不对其进行求交计算。

(3) *如何避免填充越过边界线*。边界像素的取舍问题也需要特别注意。多边形的边界与扫描线会产生两个交点，填充时如果对两个交点以及之间的区域都填充，容易造成填充范围扩大，影响最终光栅图形化显示的填充效果。为此，人们提出了“左闭右开”的原则，简单解释就是，如果扫描线交点是 1 和 9，则实际填充的区间是[1,9)，即不包括 x 坐标是 9 的那个点。

2. 扫描线算法实现

扫描线算法的整个过程都是围绕活动边表展开的，为了正确初始化活动边表，需要初始化每条扫描线的新边表，首先定义新边表的数据结构。定义新边表为一个数组，数组的每个元素存放对应扫描线的所有新边。因此定义新边表如下：

```
std::vector< std::list<EDGE> > slNet(ymax - ymin + 1);
```

ymax 和 ymin 是多边形所有顶点中 y 坐标的最大值和最小值，用于界定扫描线的范围。slNet 中

的第一个元素对应的是 ymin 所在的扫描线，以此类推，最后一个元素是 ymax 所在的扫描线。在开始对每条扫描线处理之前，需要先计算出多边形的 ymax 和 ymin 并初始化新边表。GetPolygonMinMax()函数遍历多边形的所有顶点，求出 ymax 和 ymin。

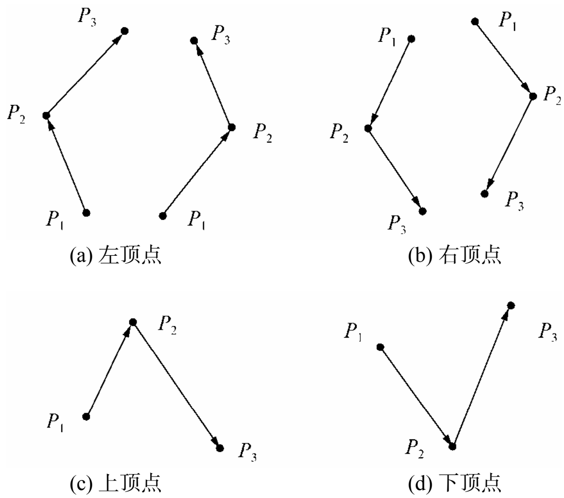


图 14-23 多边形顶点的四种类型

```
void ScanLinePolygonFill(const Polygon& py, int color)
{
    assert(py.IsValid());

    int ymin = 0;
    int ymax = 0;
    GetPolygonMinMax(py, ymin, ymax);
    std::vector< std::list<EDGE> > slNet(ymax - ymin + 1);
    InitScanLineNewEdgeTable(slNet, py, ymin, ymax);
    //PrintNewEdgeTable(slNet);
    HorizonEdgeFill(py, color); //水平边直接画线填充
    ProcessScanLineFill(slNet, ymin, ymax, color);
}
```

InitScanLineNewEdgeTable()函数根据多边形的顶点和边的情况初始化新边表，实现过程中体现了对左顶点和右顶点的区间修正原则：

```
void InitScanLineNewEdgeTable(std::vector< std::list<EDGE> >& slNet,
    const Polygon& py, int ymin, int ymax)
{
    EDGE e;
    for(int i = 0; i < py.GetPolyCount(); i++)
    {
        const Point& ps = py.pts[i];
        const Point& pe = py.pts[(i + 1) % py.GetPolyCount()];
        const Point& pss = py.pts[(i - 1 + py.GetPolyCount()) % py.GetPolyCount()];
        const Point& pee = py.pts[(i + 2) % py.GetPolyCount()];
```

```
            if(pe.y != ps.y) //不处理水平线
            {
                e.dx = double(pe.x - ps.x) / double(pe.y - ps.y);
                if(pe.y > ps.y)
                {
                    e.xi = ps.x;
                    if(pee.y >= pe.y)
                        e.ymax = pe.y - 1;
                    else
                        e.ymax = pe.y;

                    slNet[ps.y - ymin].push_front(e);
                }
                else
                {
                    e.xi = pe.x;
                    if(pss.y >= ps.y)
                        e.ymax = ps.y - 1;
                    else
                        e.ymax = ps.y;
                    slNet[pe.y - ymin].push_front(e);
                }
            }
        }
    }
```

算法通过遍历所有的顶点获得边的信息，然后根据与此边有关的前后两个顶点的情况确定此边的 ymax 是否需要-1 修正。ps 和 pe 分别是当前处理边的起点和终点，pss 是起点的前一个相邻点，pee 是终点的后一个相邻点，pss 和 pee 用于辅助判断 ps 和 pe 两个点是否是左顶点或右顶点，然后根据判断结果对此边的 ymax 进行-1 修正，算法实现非常简单，注意与扫描线平行的边是不处理的，因为水平边直接在 HorizonEdgeFill()函数中填充了。

ProcessScanLineFill()函数开始对每条扫描线进行处理，对每条扫描线的处理有四个操作，如下代码所示，四个操作分别被封装到四个函数中：

```
void ProcessScanLineFill(std::vector< std::list<EDGE> >& slNet,
    int ymin, int ymax, int color)
{
    std::list<EDGE> aet;

    for(int y = ymin; y <= ymax; y++)
    {
        InsertNetListToAet(slNet[y - ymin], aet);
        FillAetScanLine(aet, y, color);
        //删除非活动边
        RemoveNonActiveEdgeFromAet(aet, y);
        //更新活动边表中每项的 xi 值，并根据 xi 重新排序
        UpdateAndResortAet(aet);
    }
}
```

InsertNetListToAet()函数负责将扫描线对应的所有新边插入到 aet 中，插入操作到保证 aet

还是有序表，应用了插入排序的思想，实现简单，此处不多解释。FillAetScanLine()函数执行具体的填充动作，它将 aet 中的边交点成对取出组成填充区间，然后根据“左闭右开”的原则对每个区间填充，实现也很简单，此处不多解释。RemoveNonActiveEdgeFromAet()函数负责将对下一条扫描线来说已经不是“活动边”的边从 aet 中删除，删除的条件就是当前扫描线 y 与边的 ymax 相等，如果有多条边满足这个条件，则一并删除：

```
bool IsEdgeOutOfActive(EDGE e, int y)
{
    return (e.ymax == y);
}

void RemoveNonActiveEdgeFromAet(std::list<EDGE>& aet, int y)
{
    aet.remove_if(std::bind2nd(std::ptr_fun(IsEdgeOutOfActive), y));
}
```

UpdateAndResortAet()函数更新边表中每项的 xi 值，就是根据扫描线的连贯性用 dx 对其进行修正，并且根据 xi 从小到大的原则对更新后的 aet 表重新排序：

```
void UpdateAetEdgeInfo(EDGE& e)
{
    e.xi += e.dx;
}

bool EdgeXiComparator(EDGE& e1, EDGE& e2)
{
    return (e1.xi <= e2.xi);
}

void UpdateAndResortAet(std::list<EDGE>& aet)
{
    //更新 xi
    for_each(aet.begin(), aet.end(), UpdateAetEdgeInfo);
    //根据 xi 从小到大重新排序
    aet.sort(EdgeXiComparator);
}
```

其实更新完 xi 后对 aet 表的重新排序是可以避免的，只要在维护 aet 时，除了保证 xi 从小到大的排序外，在 xi 相同的情况下如果能保证修正量 dx 也是从小到大有序，就可以避免每次对 aet 进行重新排序。算法实现也很简单，只需要对 InsertNetListToAet()函数稍作修改即可，有兴趣的朋友可以自行修改。

至此，扫描线算法就介绍完了，算法的思想看似复杂，实际上并不难，从具体算法的实现就可以看出来，整个算法实现不足百行代码。

14.5.3 改进的扫描线填充算法

扫描线填充算法的原理和实现都很简单，但是因为要同时维护活动边表和新边表，对存储空间的要求比较高。这两张表的部分内容是重复的，而且新边表在很多情况下都是一张稀疏表，如

果能对其进行改进，避免出现两张表，就可以节省存储空间，同时省去从边表生成新边表的开销，同时也省去了用新边表维护活动边表的开销，基于这个原则可以对原始扫描线算法进行改进。

1. 重新设计活动边表

改进的算法仍然使用了活动边表的概念，但是不再构造独立的活动边表，而是直接在边表中划定一部分区间作为活动边区间，也就是说，把多边形的边分成两个子集，一个是与扫描线有交点的边的集合，另一个是与扫描线没有交点的边的集合。要达到这个目的，只需要对活动边表按照每条边的顶点 ymax 坐标排序即可。这个排序与原始扫描线算法中对活动边表的维护原理是一样的，因为只有边的 ymax 坐标区间内与扫描线有交点的边才可能是活动边。为了避免重复扫描整个活动边表，需要用一个 first 指针和一个 last 指针用于标识活动边区间。first 指针之前的边都是已经处理过的边，同样，last 指针之后的边都是还没有处理的边。每处理完一条扫描线，都要更新 first 和 last 指针位置，调整 last 指针的位置将 ymax 大于当前扫描线的边纳入到活动边区间，同时调整 first 指针将处理完成的边排除在活动边区间之外。

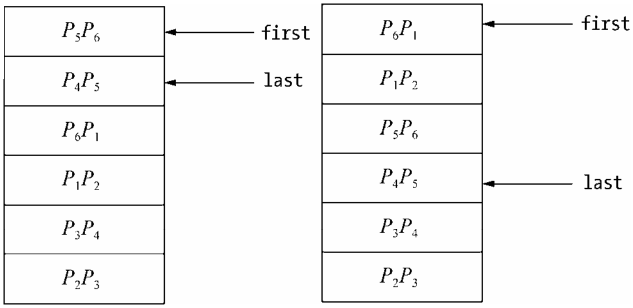
如果调整 last 指针的依据是边的 ymax 是否大于当前扫描线，那么调整 first 指针的依据是什么？也就是如何判断一条边已经处理完了？方法是在边（EDGE）定义中增加一个 dy(Δy)属性，这个属性被初始化成这条边在 y 方向上的长度，每处理完一条扫描线，dy 都要做减一处理，当 dy = 0 时，就说明这条边已经不与扫描线相交了，可以被排除在活动边区间之外。改进的扫描线算法的“边”的完整定义如下：

```
typedef struct tagEDGE2
{
    double xi;
    double dx;
    int ymax;
    int dy;
}EDGE2;
```

EDGE2 定义中 xi、dx 和 ymax 的含义和原始算法中 EDGE 的定义相同，只是多了一个 dy 属性。

每当处理一条扫描线时，除了活动边区间的 first 指针和 last 指针需要调整之外，还要将 first 指针和 last 指针之间的活动边按照 xi 从小到大的顺序排序，以保证填充算法能够用正确的交点线段序列画线填充。因此，每次调整活动边区间的 first 指针和 last 指针之后，都要对活动边区间重新排序，也就是说活动边区间内的各边的位置并不固定，会随着扫描线的变化而相应地变化。

仍以图 14-20 所示的多边形为例，处理扫描线 10 时的活动边表状态如图 14-24a 所示，而处理扫描线 8 时的活动边表状态则如图 14-24b 所示。可以看出，当处理扫描线 8 时，活动边区间内的边的顺序有了调整，因为新加入的 P_6P_1 和 P_1P_2 两条边与扫描线的交点坐标 x_i 比 P_5P_6 与扫描线的交点坐标 x_i 小，因此排在 P_5P_6 前面。



(a)　　(b)

图 14-24　改进的活动边表结构

2. 新活动边表的构造与调整

改进的扫描线算法的重点是活动边表的构造和调整，构造方法如下。

- 首先剔除多边形各边的水平边，然后将剩下的边按照 ymax 的值从大到小的顺序存入一个线性表中，表中第一个元素是 ymax 值最大的表，最后一个元素是 ymax 值最小的边。对于各边中左、右顶点的情况需要和原始算法一样做调整，以免出现交点个数不正确的异常。这里对调整的策略再强调一下，调整都是针对边的终点进行的，对于图 14-23a 所示的左顶点，需要先将 P_2 点的坐标调整为$(x_2 - dx, y_2 - 1)$，然后再求边的 ymax、x_i 和 dy。对于图 14-23b 所示的右顶点，需要将 P_2 点的坐标调整为$(x_2 + dx, y_2 + 1)$，然后再求边的 ymax、x_i 和 dy。
- 加入 first 指针和 last 指针，构成活动边区间。first 指针和 last 指针之间的边都是和当前扫描线有交点的边或已经处理过的边，已经处理过的边的 dy 是 0，因此，对活动边扫描时需要忽略其中 dy 已经是 0 的边。这些已经处理过的边会加载在正常的边中，直到调整 first 指针时被剔除出活动边区间。

活动边表的调整指的是在处理完每根扫描线之后，更新活动边表中活动边区间内的各边的相关属性的值，比如递减 dy 的值，调整交点 x_i 坐标的值等。根据 EDGE2 的定义，每根扫描线处理完之后需要对活动边区间内的边做两步调整，首先调整活动边区间中参与求交计算的各边的属性值，这些调整算法是：

```
dy = dy - 1;
xi = xi - dx;
```

然后调整活动边区间的 first 指针和 last 指针，使符合条件的新边加入到活动边区间，同时将处理完的边从活动边区间剔除。这些调整算法是：

```
if(first 所指边的Δy 为 0)
    first=first+1;
if(last 所指的下一条边的 ymax 大于下一扫描线的 y 值)
    last=last+1
```

3. 改进的扫描线填充算法实现

首先定义活动边表，这是一个线性表，每个元素是一条边的全部属性，同时还要包含 first 指针和 last 指针，其数据结构定义如下：

```
typedef struct tagSP_EDGES_TABLE
{
    std::vector<EDGE2> slEdges;
    int first;
    int last;
}SP_EDGES_TABLE;
```

改进的扫描线填充算法重点仍然是新活动边表的构造，构造新活动边表的算法实现如下：

```
void InitScanLineEdgesTable(SP_EDGES_TABLE& spET, const Polygon& py)
{
    EDGE2 e;
    for(int i = 0; i < py.GetPolyCount(); i++)
    {
        const Point& ps = py.pts[i];
        const Point& pe = py.pts[(i + 1) % py.GetPolyCount()];
        const Point& pee = py.pts[(i + 2) % py.GetPolyCount()];

        if(pe.y != ps.y) //不处理水平线
        {
            e.dx = double(pe.x - ps.x) / double(pe.y - ps.y);
            if(pe.y > ps.y)
            {
                if(pe.y < pee.y) //左顶点
                {
                    e.xi = pe.x - e.dx;
                    e.ymax = pe.y - 1;
                    e.dy = e.ymax - ps.y + 1;
                }
                else
                {
                    e.xi = pe.x;
                    e.ymax = pe.y;
                    e.dy = pe.y - ps.y + 1;
                }
            }
            else //(pe.y < ps.y)
            {
                if(pe.y > pee.y) //右顶点
                {
                    e.xi = ps.x;
                    e.ymax = ps.y;
                    e.dy = ps.y - (pe.y + 1) + 1;
                }
                else
                {
                    e.xi = ps.x;
                    e.ymax = ps.y;
                    e.dy = ps.y - pe.y + 1;
```

```
                }
            }

            InsertEdgeToEdgesTable(e, spET.slEdges);
        }
    }
    spET.first = spET.last = 0;
}
```

多边形 Polygon 中的 pts 数组按照顺序存放了多边形的各个顶点，InitScanLineEdgesTable()函数从 pts 中依次取出三个顶点，前两个顶点构成当前处理的边，后一个顶点用于辅助判断是否是左、右顶点的情况，如果是左、右顶点的情况，就要对边的终点的坐标做调整（调整的方法我们在前面已经描述过）。调整完线段终点坐标后构造边 e，然后由 InsertEdgeToEdgesTable()函数将 e 插入到线性表中,插入操作满足线性表按照 ymax 从大到小排序,这个是插入排序的基本算法，这里就不再列出代码。

算法的另一个重点就是处理每条扫描线和活动边表的关系,计算出每条扫描线需要填充的区间。这个算法体现在 ProcessScanLineFill2()函数中：

```
void ProcessScanLineFill2(SP_EDGES_TABLE& spET,
    int ymin, int ymax, int color)
{
    for (int yScan = ymax; yScan >= ymin; yScan--)
    {
        UpdateEdgesTableActiveRange(spET, yScan);
        SortActiveRangeByX(spET);
        FillActiveRangeScanLine(spET, yScan, color);
        UpdateActiveRangeIntersection(spET);
    }
}
```

ProcessScanLineFill2()函数依次处理每条扫描线，根据 14.5.3 第 2 小节的算法描述，UpdateEdges TableActiveRange()函数和 SortActiveRangeByX()函数更新活动边区间并对区间内的边排序，FillActiveRangeScanLine 函数从活动边区间内依次取出两个交点组成填充区间，调用前面介绍的 DrawHorizontalLine()函数完成画线填充，UpdateActiveRangeIntersection()函数则根据 14.5.3 第 2 小节的算法描述更新参与求交计算的各边的属性值。这四个函数的实现都很简单，结合 14.5.3 第 2 小节的算法描述很容易理解。

14.5.4 边界标志填充算法

在光栅显示平面上，多边形是封闭的，它是用某一边界色围成的一个闭合区域，填充是逐行进行的，即用扫描线逐行对多边形求交，在交点对之间填充。边界标志填充算法就是在逐行处理时，利用边界或边界颜色作为标志来进行填充。准确地说，边界标志填充算法不是指某种具体的填充算法，而是一类利用扫描线连贯性思想的填充算法的总称。这类算法有很多种，本节就介绍两种常见的边填充算法。

1. 以边为中心的填充算法

首先介绍一种以边为中心的边缘填充算法，这种边界标志算法的基本思想是：对于每一条扫描线和每一条多边形边的交点(x_i,y_i)，将该扫描线上交点右方的所有像素取补，依次对多边形的每条边做此处理，直到最终完成填充。这里要介绍一下取补的定义，假设某点的颜色是 M，则对该点的颜色取补得到 $M' = A - M$，A 是一个很大的数字，至少要比所有合法的颜色值大。根据取补的定义，如果对光栅位图某区域已经标记为 M 的颜色值做偶数次取补运算，该区域颜色不变；而做奇数次取补运算，则该区域颜色变为值为 M' 的颜色。算法的处理过程可以简单地描述为以下两个步骤。

(1) 将绘图窗口的背景色置为 M' 颜色；

(2) 对多边形的每一条非水平边，从该边上的每个像素开始向右求余。

图 14-25 展示了这两个步骤的处理流程，左边是多边形的形状，右边分别是对每条边处理完成后填充区域的颜色情况，初始背景颜色是 M'，经过处理后，需要填充的区域是奇数次取补，最终的颜色是要填充的正确值 M，非填充区域经过偶数次取补，仍然是背景色 M'。

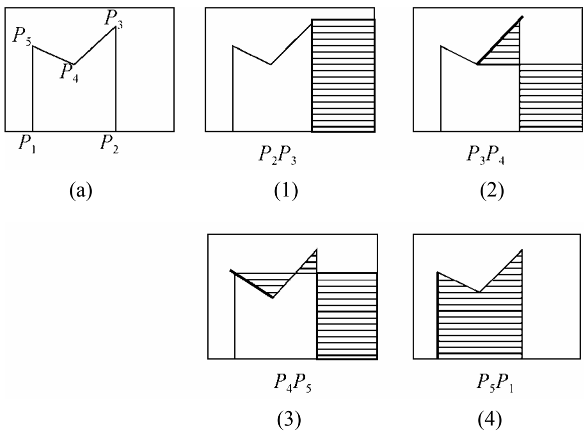


图 14-25　边缘填充算法的处理过程

算法的实现非常简单，对于光栅位图的展示，我们仍然采用之前所用的方法，用数字矩阵表示一块光栅位图区域，矩阵的每个位置表示一个像素点，用 0～9 表示颜色值。本算法示例用 9 表示最大值 A，0 表示无效的区域，合法的颜色值就是 1～8。

```
void EdgeCenterMarkFill(const Polygon& py, int color)
{
    std::vector<EDGE3> et;

    InitScanLineEdgesTable(et, py);//初始化边表

    FillBackground(A - color); //对整个填充区域背景颜色取补
    for_each(et.begin(), et.end(), EdgeScanMarkColor);//依次处理每一条边
```

```
}
void EdgeScanMarkColor(EDGE3& e)
{
    for(int y = e.ymax; y >= e.ymin; y--)
    {
        int x = ROUND_INT(e.xi);
        ComplementScanLineColor(x, MAX_X_CORD, y);

        e.xi -= e.dx;
    }
}
```

InitScanLineEdgesTable()函数前面已经介绍过，FillBackground()函数将填充背景初始化为要填充颜色的取补颜色，EdgeScanMarkColor()函数负责对每条非水平边进行处理，逐条扫描线进行颜色取补，ComplementScanLineColor()函数负责对 y 扫描线上$[x_1, x_2]$区间的点的颜色值取补。

2. 栅栏填充算法

以边为中心的填充算法的优点是简单,缺点是对于复杂多边形,每一像素可能被访问多次(多次取补)，效率不高。考虑对此算法改进，人们提出了*栅栏填充算法*。栅栏填充算法的基本思想是：经过多边形的某个顶点，在多边形内部建立一个与扫描线垂直的“栅栏”，当扫描线与多边形边有交点时，就将交点与栅栏之间的像素取补。若交点位于栅栏左边，则将交点之右，栅栏之左的所有像素取补；若交点位于栅栏右边，则将栅栏之右，交点之左的像素取补。

仍以上一节介绍的多边形为例，假设经过 P_4点建立一条栅栏，则改进的栅栏填充算法处理过程就如图 14-26 所示。栅栏填充算法的实现和以边为中心的边缘填充算法类似，只是对每条边的扫描线取补处理的范围控制有区别，这就是算法需要指定一个“栅栏”的原因。注意本算法中FenceScanMarkColor()函数和 EdgeScanMarkColor()函数的区别,就是这点区别使得栅栏填充算法主动减少了很多像素被访问的次数，而多边形之外的像素也不会被多余处理，效率提高了不少。

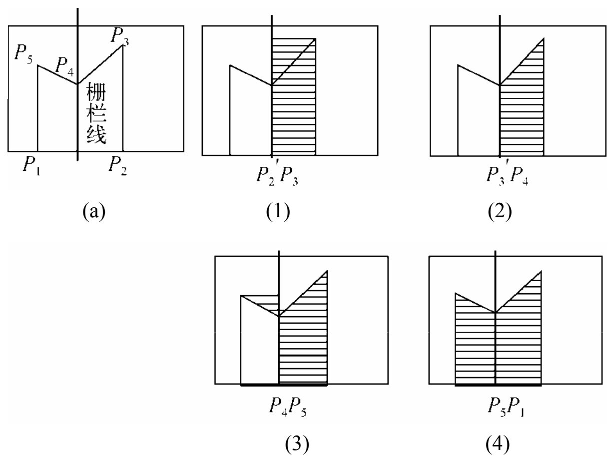


图 14-26　栅栏填充算法的处理过程

```
void EdgeFenceMarkFill(const Polygon& py, int fence, int color)
{
    std::vector<EDGE3> et;

    InitScanLineEdgesTable(et, py);//初始化边表

    FillBackground(A - color); //对整个填充区域背景颜色取补
    for_each(et.begin(), et.end(),
        std::bind2nd(std::ptr_fun(FenceScanMarkColor), fence));//依次处理每一条边
}

void FenceScanMarkColor(EDGE3 e, int fence)
{
    for(int y = e.ymax; y >= e.ymin; y--)
    {
        int x = ROUND_INT(e.xi);
        if(x > fence)
        {
            ComplementScanLineColor(fence, x, y);
        }
        else
        {
            ComplementScanLineColor(x, fence - 1, y);
        }

        e.xi -= e.dx;
    }
}
```

14.6 总结

本章介绍了计算机图形学中一些常见的算法，还有包括矢量在内的一些计算几何的知识。这些都是最基础的内容，但是通过对这些内容的了解，你可以打开算法世界的一个重要分支的大门。比如点与多边形的关系、判断多边形的凸凹性以及判断多边形之间的相交关系，都是算法比赛中比较常见的题目。

14.7 参考资料

[1] 周培德. 计算几何：算法设计与分析. 北京：清华大学出版社，2005

[2] 德贝尔赫. 计算几何：算法与应用. 邓俊辉译. 北京：清华大学出版社，2005

[3] 孙家广，杨常贵. 计算机图形学. 北京：清华大学出版社，1995

[4] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[5] 同济大学数学系. 高等数学（第六版）. 北京：高等教育出版社，2007

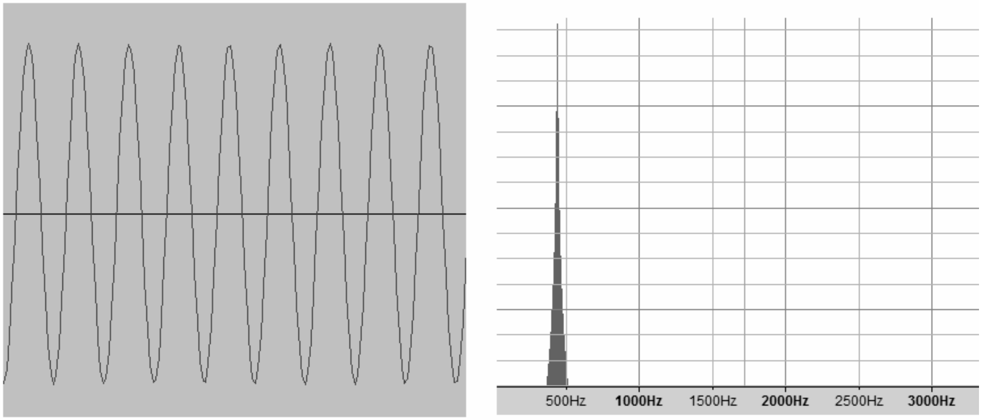
第 15 章 音频频谱和均衡器与傅里叶变换算法

音频播放时的频谱实时显示和调整音效的均衡器功能是各种媒体播放程序的标准配置，小伙伴们可曾疑惑过它们的实现原理？我上学的时候也想自己编程做一个 MP3 播放器，对 Winamp 界面上那个跳动的频谱十分向往，最终因为不知道实现原理而放弃了。后来学了《数值分析》，才知道原来背后的原理就是傅里叶变换算法。本章就介绍一下傅里叶变换算法如何应用到音频播放的频谱和均衡器的实现中，同时顺带介绍一下根据电话拨号音破解电话号码的小把戏，当然，这也离不开傅里叶变换算法。

15.1 实时频谱显示的原理

频谱实际上是信号分析领域里的一个专属概念，是一段音频（或图像）数据在频域内的表示。频域是相对于时域的一个概念，在时域内的信号，其坐标轴是时间轴，时域信号表示信号强度随时间变化的情况。在频域内的信号，其坐标轴是频率，频域信号表示的是信号在各个频率上的相对功率强度。

很多情况下，信号的某些特征在时域内表现得并不明显，但是如果转换到频域，则相应的特征就一目了然了。将时域信号转换成频域信号，是信号分析领域里一种常用的方法。下面就以 440Hz 的正弦波为例，通过其在时域和频域内的图像展示，理解一下这种转换的意义。图 15-1a 是 440Hz 的正弦波在时域内的形态，音频采样率是 8000Hz。图 15-1b 是其在频域内的形态，理想状态下，图 15-1b 应该在 440Hz 处显示一条直线，其他位置的值都是 0，但是受原始信号杂波和转换后的频域分辨率影响，实际显示的是一个呈金字塔形状的图形，不过还是可以明显地看到在 440Hz 的时候功率（相对强度）值最大，其他位置的值都明显小于 440Hz 位置的值。



(a) (b)

图 15-1 440Hz 正弦波在时域和频域内的形态

媒体播放程序中实时显示的频谱之所以和正在播放的音乐匹配，是因为它反映的就是当前正在播放的一小段音频在各个频段上的强度。例如 1 秒钟的音频数据如果分成 5 个 Buffer 在回放设备上播放，每播放完一个 Buffer 后，将这个 Buffer 的音频数据（通常是时域信号）转换成频域数据，统计出这段数据在各个频段上的强度，然后在频谱窗口中以图形化的方式体现出来。这样 1 秒钟就可以刷新 5 次，形成跳动的频谱，并与当前播放的声音形成互动，这就是实时显示频谱的原理。

原理一点都不复杂，但是怎么将音频数据从时域转换到频域呢？现在该大名鼎鼎的离散傅里叶变换（Discrete Fourier Transform，DFT）隆重登场了！下一节就介绍离散傅里叶变换的推导原理和算法实现。

15.2 离散傅里叶变换

在数字信号分析领域里，将信号转换成频域信号的方法很多，傅里叶变换是其中最常用的一种。傅里叶变换算法实现简单，特别是 J. W. 库利和 T. W. 图基在 1965 年提出了快速傅里叶变换算法（FFT），更是将傅里叶变换的速度提高了千万倍。快速傅里叶变换算法极大地减少了算法的计算量，推动傅里叶变换算法在各个领域得到了广泛的应用。

传统的傅里叶变换都是连续函数，用于处理无限长度的连续周期性时域信号，但是不适用于计算机实现。计算机受存储器的限制不能处理无限长度的连续信号，只能一次一批地处理有限长度的离散信号，这就需要对信号进行离散化处理，同时建立对应的离散信号傅里叶变换。对离散信号进行傅里叶变换的方法就是离散傅里叶变换。下面就来介绍一下离散傅里叶变换的前世今生，以及快速傅里叶变换的原理和算法实现，这些是本章要介绍的各种应用的基础。

15.2.1　什么是傅里叶变换

傅里叶是一位法国数学家和物理学家的名字，他于 1807 年在法国科学学会上发表了一篇论文，提出了一个观点：任何连续周期信号可以由一组适当的正弦曲线组合而成，换言之，满足一定条件的连续函数（周期函数）都可以表示成一系列三角函数（正弦和/或余弦函数）或者它们的积分的线性组合形式，这个转换就称为傅里叶转换。一般情况下，若“傅里叶变换”一词的前面未加任何限定语，则指的是“连续傅里叶变换”，与之对应的自然就是“离散傅里叶变换”。离散傅里叶变换其实可以看作是“离散时间傅里叶变换”（Discrete-Time Fourier Transform，DTFT）的一个特例，离散时间傅里叶变换在时域是离散的，但是在频域是连续的，而离散傅里叶变换则在时域和频域都以离散的形式呈现，因此，离散傅里叶变换更适用于所有使用计算机处理数据的场合。

离散傅里叶变换需要对原始的连续信号进行离散化，原始信号离散化的过程其实就是以一定的采样周期对原始信号进行采样的过程。最典型的例子就是脉冲编码调制（Pluse Code Modulation，PCM）技术对音频信号的处理方式，连续的声音信号（模拟信号），通过采样，变成一个一个采样数据（数字信号）。如果采样周期是 8000Hz，则一秒钟的的声音会变成 8000 个采样数据。计算机系统回放设备播放的声音数据就是使用各种采样周期得到的离散化的 PCM 音频，频谱和均衡器也都是基于离散化的 PCM 数据进行处理的。

15.2.2　傅里叶变换原理

要了解离散傅里叶变换的原理，首先要从连续傅里叶变换开始。因为工程中用的最广泛的是非周期信号，所以我们只关注非周期信号的处理方式。非周期信号傅里叶变换的基本思想就是：把非周期信号当成一个周期无限大的周期信号，然后研究这个无限大周期信号的傅里叶转换的极限特征。换句话说，就是把你要处理的非周期信号看作是一个只有一个周期的周期信号，所有的数据是周期性的，只不过只有一个周期而已。

1. 离散傅里叶变换公式

先来看看连续非周期信号的傅里叶变换公式：

$$X(\omega)=\int_{-\infty}^{\infty}x(t)\mathrm{e}^{-\mathrm{i}\omega t}\mathrm{d}t \tag{15-1}$$

公式中的 i 是虚数单位，即 $\mathrm{i}^2=-1$。接下来要对连续傅里叶变换离散化，连续傅里叶变换中的函数 $x(t)$是连续的，现在假设在 $x(t)$的某一段连续区间上以周期 T 进行采样，得到 N 个采样点，则每个采样点的离散傅里叶变换公式就是：

$$X(n)=\sum_{k=0}^{N-1}x(k)\mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}kn}\qquad n=0,1,\cdots,N-1 \tag{15-2}$$

积分变成了级数求和，这就是离散化的结果。如果要计算这段区间上 $x(t)$处的傅里叶转换结

果，就可以通过计算离散信号的 $x(nT)$获得。考察式(15-2)可知，计算 N个采样点的傅里叶变换，需要计算 N^2次复数乘法运算和 $N(N-1)$次复数加法运算。式(15-2)中 e 的指数项 $\mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}}$是个与点数 N有关的常量，令 $W_N=\mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}}$，则式(15-2)可简单记为式(15-3) 所示：

$$X(n)=\sum_{k=0}^{N-1}x(k)W_N^{nk} \qquad n=0,1,\cdots,N-1 \tag{15-3}$$

2. 快速傅里叶变换原理推导

由于 DFT 算法计算量巨大，限制了 DFT 的应用。长期以来，人们提出了很多改进的离散傅里叶变换算法，其中，J. W. 库利和 T. W. 图基发明的快速傅里叶变换（Fast Fourier Transform，FFT）算法就是使用最广泛的一种。在开始理解快速傅里叶变换之前，先了解一下式(15-3)中自然对数 e 的指数项 W_N的周期性和对称性，W_N的周期性可以表示为：

$$W_N^{nk}=W_N^{(N+n)k}=W_N^{n(N+k)} \tag{15-4}$$

W_N的对称性可以表示为：

$$W_N^{nk}=W_N^{-nk}=W_N^{(N-n)k}=W_N^{n(N-k)} \tag{15-5}$$

快速傅里叶变换算法的基本思想就是利用以上周期性和对称性，将 N个点的 DFT 分解成 n个 N/n点的 DFT，从而显著地减少了运算量。事实上，这个想法并不是库利和图基两个人的首创，大数学家高斯在 1805 年就发明了这种算法的基本思想，不过基 2 的快速傅里叶变换算法仍然以这两个人的名字命名，以表彰他们对推广傅里叶变换应用所作的贡献。有一点值得说一下，这种算法的思想是如此优秀，以至于库利和图基并不是历史上第一个重复发明它的人。也许是“英雄所见略同”的缘故，此算法在历史上不断被各个研究领域的学者们“重复发明”。

现在就以基 2 的 FFT 算法为例，介绍一下这种分解如何有效地减少运算量。如图 15-2 所示，将 N个点的 DFT 分解成两个 $N/2$个点的 DFT，可以将复数乘法的运算量减少为 $N^2/2$。再进一步分解成四个 $N/4$个点的 DFT，复数乘法的计算量进一步减少为 $N^2/4$。分解过程可以迭代进行，直到不能再分解为止（2 个点的 DFT）。

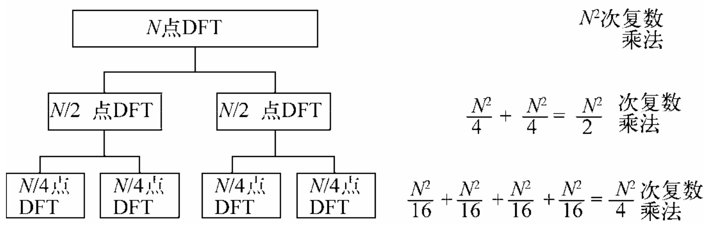


图 15-2 基 2 FFT 分解与计算量

每次分解，都将原始信号 $x(n)$按照时间顺序（也就是 n 的序号）分成奇、偶两个组 $x_1(r)$和 $x_2(r)$，其中 r 与 n 的关系是：

当 n 为偶数时，令 $n=2r$；

当 n 为奇数是，令 $n=2r+1$；

也就是说，原始信号与两个分组的信号存在以下关系：

$$x(2r)=x_1(r),\quad x(2r+1)=x_2(r)\qquad 其中\ r=0,1,\cdots,N/2-1 \tag{15-6}$$

现在将式(15-6)中的关系代入到式(15-3)中，将一个 N 点 DFT 分解为两个 $N/2$ 点 DFT：

$$X(n)=\sum_{k=0}^{N-1}x(k)W_N^{nk}=\sum_{r=0}^{N/2-1}x(2r)W_N^{2rn}+\sum_{r=0}^{N/2-1}x(2r+1)W_N^{(2r+1)n}$$

$$=\sum_{r=0}^{N/2-1}x_1(r)W_N^{2rn}+W_N^n\sum_{r=0}^{N/2-1}x_2(r)W_N^{2rn} \tag{15-7}$$

因为 $W_N^{2r}=\mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}2r}=\mathrm{e}^{-\mathrm{i}\frac{2\pi}{N/2}r}=W_{N/2}^r$，将此递推关系代入式(15-7)，得到：

$$X(n)=\sum_{r=0}^{N/2-1}x_1(r)W_{N/2}^{rn}+W_N^n\sum_{r=0}^{N/2-1}x_2(r)W_{N/2}^{rn}=X_1(n)+W_N^nX_2(n) \tag{15-8}$$

式(15-8)中，N 点 DFT 变换 $X(n)$中 n 的取值范围是 $0,1\cdots,N-1$，周期为 N，而两个 $N/2$ 点 DFT 变换 $X_1(n)$和 $X_2(n)$中 n 的取值范围是 $0,1\cdots,N/2-1$，周期为 $N/2$。因此，式(15-8)只是给出了 $N/2$ 点的变换关系，并没有将全部 N 个点的 $X(n)$中都求解出来。要想利用 $X_1(n)$和 $X_2(n)$表达全部的 $X(n)$中，还必须利用 W_N的周期性和对称性进，找出 $X_1(n)$、$X_2(n)$和 $X(n+N/2)$的关系，进一步推导出后 $N/2$ 个点的对应关系。由式(15-4)的周期性可知：$W_{N/2}^{r(N/2+n)}=W_{N/2}^{rn}$，因此：

$$X_1(N/2+n)=\sum_{r=0}^{N/2-1}x_1(r)W_{N/2}^{r(N/2+n)}=\sum_{r=0}^{N/2-1}x_1(r)W_{N/2}^{rn}=X_1(n)$$

同理可得：$X_2(N/2+n)=X_2(n)$。

由式(15-8)推导的分解关系可知，一个 N 点 DFT 变换的前 $N/2$ 个周期数据的转换关系是：

$$X(n)=X_1(n)+W_N^nX_2(n)\qquad n=0,1,\cdots,N/2-1 \tag{15-9}$$

其后 $N/2$ 个周期数据的转换关系是：

$$X(N/2+n)=X_1(N/2+n)+W_N^{N/2+n}X_2(N/2+n)=X_1(n)+W_N^{N/2+n}X_2(n)$$

由式(15-5)中 W_N^n 的对称性可知，$W_N^{N/2+n}=W_N^{N/2}\cdot W_N^n=-W_N^n$，代入上式后得到后 $N/2$ 个周期数据的转换关系：

$$X(N/2+n)=X_1(n)-W_N^nX_2(n)\qquad n=0,1,\cdots,N/2-1 \tag{15-10}$$

由式(15-9)和式(15-10)可知，N 点 DFT 变换 $X(n)$分解的 $X_1(n)$和 $X_2(n)$表，通过如图 15-3 所示的蝶形运算关系，建立与 $X(n)$的每个点的映射关系。

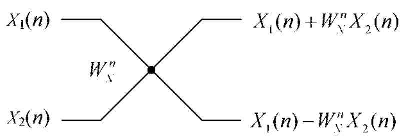


图 15-3 蝶形运算关系图

3. 蝶形运算后的码位倒序关系

由上一节的快速傅里叶变换原理推导过程可知，FFT 算法的每一级迭代计算，都是由 N 个输入数据（复数）两两分组构成 $N/2$ 个蝶形运算，经过蝶形运算后得到 N 个输出数据（复数）。但是，经过蝶形运算之后，每个数据的位置都发生了变化。以 8 点 FFT 运算为例，图 15-4 显示了蝶形运算后 8 个点的数据的位置关系。从图中可以看出，为了保证蝶形运算后输出的顺序与原始序列一致，在进行蝶形运算之前，需要对原始序列进行重新排序。FFT 算法对这个位置关系的处理有两种方式，一种如图 15-4 所示，在开始蝶形运算之前就对原始数据按照码位关系进行排序，则计算后可直接得到与原始序列一致的输出，这种方式又称为原位运算方式。另一种方式是直接进行蝶形运算，然后按照码位关系对运算后的输出结果重新排序。

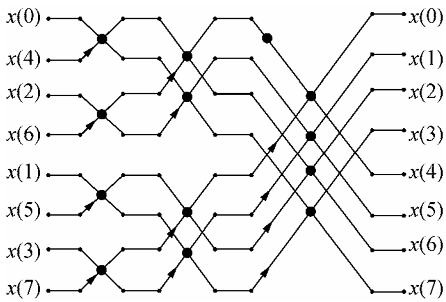


图 15-4 8 点 FFT 蝶形运算位置关系图

蝶形运算的码位关系看起来相当杂乱，然而还是有一定的规律的，这个规律就是码位倒读规律。仍以 8 点 FFT 为例，表 15-1 显示了这种倒读规律。

表15-1 码位倒序关系表

原始顺序	原始顺序二进制码	倒读后的二进制码	码位倒读顺序
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

由此可知，原始数据序列中的某个数据，经过 FFT 的蝶形运算后的位置，可以通过这个数据在原始数据序列中的序号，经过二进制反序后得到。同理，经过 FFT 的蝶形运算后的某个数据，也可以根据在转换后的数据序列中的序号，经过二进制反序后得到对应的数据在原始数据序列中的位置。

15.2.3 快速傅里叶变换算法的实现

经过上一节的分析，看似神秘的快速傅里叶变换算法其实非常简单。算法的实现以分治法为策略，递归地将长度为 N 的 DFT 分解为两个长度为 $N/2$ 的 DFT。假如转换数据的点数 N 是 2 整数幂，可表示为 $N=2^M$，则算法需要进行 M 阶迭代分解，每一阶都有 $N/2$ 个蝶形运算。

考察图 15-3 的蝶形运算关系图，每次蝶形运算都要乘以因子 W_N^n，而且每次蝶形运算都会使得两个原始数据的结果像被扭转了一样变换位置，因此这个因子又称为旋转因子。整个 FFT 算法的每一阶迭代分解中，旋转因子的个数是不一样的。第一阶分解为两个 $N/2$ 的 DFT，有一个旋转因子，第二阶再分解为四个 $N/4$ 的 DFT，有两个旋转因子，以此类推。第 L 阶迭代分解运算的旋转因子指数 n 的计算方法是：

$$n = \mathrm{j}\cdot 2^{M-L} \quad （\mathrm{j}\text{ 是 } L \text{ 阶的第 j 个旋转因子}） \tag{15-11}$$

旋转因子 W_N^n 是个复指数，在算法实现时通常要分解成正弦和余弦函数的分解形式，对于 W_N^n 的分解可表示为：

$$W_N^n = \mathrm{e}^{-\mathrm{i}\frac{2n\pi}{N}} = \cos(\frac{2n\pi}{N}) - \mathrm{i}\sin(\frac{2n\pi}{N}) \tag{15-12}$$

很多 FFT 算法的实现通常事先计算好分解式中的正弦项和余弦项，存放在一张数据表中，在进行蝶形运算的过程中，通过查表可以直接获取事先计算好的值，不必每次都计算，提高了算法效率。本章要给出的快速傅里叶变换算法也采用了这种方法，在 `InitFft()`函数中预先计算好正弦项和余弦项的值。

至此，FFT 算法的全部原理和细节都已经介绍完毕，算法实现已经不存在任何技术问题，这里给出一个中规中矩的算法实现，完全对照本章的推导过程实现。工业上有很多更高效的算法实现，有需要的读者可以直接研究它们。

```
void FFT(FFT_HANDLE *hfft, COMPLEX *TD2FD)
{
    int i,j,k,butterfly,p;

    int power = NumberOfBits(hfft->count);

    /*蝶形运算*/
    for(k = 0; k < power; k++)
    {
        for(j = 0; j < 1<<k; j++)
```

```
        {
            butterfly = 1 << (power-k);
            p = j * butterfly;
            int s = p + butterfly / 2;
            for(i = 0; i < butterfly/2; i++)
            {
                COMPLEX t = TD2FD[i + p] + TD2FD[i + s];
                TD2FD[i + s] = (TD2FD[i + p] - TD2FD[i + s]) * hfft->wt[i*(1<<k)];
                TD2FD[i + p] = t;
            }
        }
    }

    /*重新排序*/
      for (k = 0; k < hfft->count; k++)
    {
         int r = BitReverise(k, power);
         if (r > k)
         {
            COMPLEX t = TD2FD[k];
            TD2FD[k] = TD2FD[r];
            TD2FD[r] = t;
         }
    }
}
```

离散傅里叶转换算法是基于复数的，参数 TD2FD 是个复数数组，COMPLEX 的定义如下：

```
struct COMPLEX
{
    float re;
    float im;
};
```

NumberOfBits()函数根据点数 N 计算出 2 的整数幂 M，BitReverise()函数实现码序翻转。重新排序的过程中，规定只有反序后的码序比当前值大才交换位置，避免了重复交换码序。FFT_HANDLE 的初始化主要是计算前面介绍的正弦和余弦表和窗口函数表，关于窗口函数表，下一节介绍频谱应用的时候再介绍，这里只是给出实现代码：

```
bool InitFft(FFT_HANDLE *hfft, int count, int window)
{
    int i;

    hfft->count = count;
    hfft->win = new float[count];
    if(hfft->win == NULL)
    {
        return false;
    }
    hfft->wt = new COMPLEX[count];
    if(hfft->wt == NULL)
    {
        delete[] hfft->win;
```

```
        return false;
    }
    for(i = 0; i < count; i++)
    {
        hfft->win[i] = float(0.50 - 0.50 * cos(2 * M_PI * i / (count - 1)));
    }
    for(i = 0; i < count; i++)
    {
        float angle = -i * M_PI * 2 / count;
        hfft->wt[i].re = cos(angle);
        hfft->wt[i].im = sin(angle);
    }

    return true;
}
```


15.3 傅里叶变换与音频播放的实时频谱显示

一次独立的离散傅里叶转换只能将有限个数的数据转换到频域，如果一段音频数据比较长，需要进行多次傅里叶变换，那么就需要对转换后的频域数据进行叠加，才能计算出每个频率上总的功率，也就是频谱。所以，在进行频谱计算之前，我们首先介绍一下时域信号转换成频域信号后有哪些特点，以及如何利用频域数据进行分析和计算。

15.3.1 频域数值的特点分析

图 15-1 给大家一个频域数据的抽象认识，本节将帮助大家更加具体地认识频域数据。前面提到过，原始信号离散化的过程其实就是以一定的周期对原始信号进行采样的过程，这里就提到了一个很重要的参数，就是采样率 T。还有一个很重要的参数，就是每次进行转换的时域信号的个数 N（也称为离散傅里叶变换的点数），因为这两个参数共同决定了转换后频域数值的频域分辨率。

时域数据显示了音频信号强度随时间变化的趋势，横坐标轴就是时间，纵坐标轴就是信号强度。时间坐标轴的分辨率由采样率 T 决定，其值为 $1/T$（$1/T$ 就是采样周期）。频域信号显示了音频信号强度随频率变化的趋势，横坐标轴是频率，纵坐标轴是信号强度（功率）。频率坐标轴的分辨率由时域采样率 T 和离散傅里叶转换点数 N 共同决定，其值为 T/N。因为离散傅里叶转换将时域信号一对一的转换为频域信号，也就是说，N 个时域信号转换后会得到 N 个频域信号，这 N 个频域信号对应的频率范围是 $0–T$，所以每个频域信号的对应的频段宽度是 T/N。

离散傅里叶转换得到 N 个频域数据，第一个点对应的是频率为 0Hz 的信号强度，也就是音频数据中直流信号的强度。第二个点是频率 T/N Hz 对应的信号强度，以此类推，第 n 个点对应的是频率为$(n-1)T/N$ Hz 的信号强度。此外，频域数据还有一个特点，就是对称性，其前 $N/2$ 个点的数值和后 $N/2$ 个点的数值呈现轴对称特性，所以在计算功率谱时，只考虑前 $N/2$ 个点就可以了。

15.3.2　从音频数据到功率频谱

进行傅里叶转换之前，要先对音频数据进行处理，将其转化为算法支持的数据格式。傅里叶转换是针对复数的，算法实现也使用了复数，但是音频数据是实数，因此，转化成复数时只使用实部，虚部为 0。

音频数据通常来自与文件或声音采集设备，常用的音频信号格式是 PCM 编码。WAVE 文件就是最常见的音频文件格式，其数据采用的就是 PCM 编码，数据以一个一个采样点顺序存放的方式存储，转换时只要逐个对采样数据进行转换即可。如果音频数据包含多个声轨，比如双声道立体声模式就包含左、右两个声轨的数据，这种情况只需要计算多个声轨的平均值作为一个采样数据。具体的转换算法如下：

```
void SampleDataToComplex(short *sampleData, int channels, COMPLEX *cd)
{
    if(channels == 1)
    {
        cd->re = float(*sampleData / 32768.0);
        cd->im = 0.0;
    }
    else
    {
        cd->re = float(*sampleData + *(sampleData + 1) / 65536.0);
        cd->im = 0.0;
    }
}
```

1. 窗函数与窗口滑动

计算机不能处理无限长度的数据，离散傅里叶变换算法只能对数据一批一批地进行变换，每次只能对限时间长度的信号片段进行分析。具体的做法就是从信号中截取一段时间的片段，然后对这个片段的信号数据进行周期延拓处理，得到虚拟的无限长度的信号，再对这个虚拟的无限长度信号进行傅里叶变换。但是信号被按照时间片截取成片段后，其频谱就会发生畸变，这种情况也称为频谱能量泄露。

为了减少能量泄露，人们研究了很多截断函数对信号进行截取操作，这些截断函数称为窗函数。窗函数 $w(t)$被设计成频带无限的函数，所以即使原始信号是有限带宽信号，被窗函数截取后得到的片段也会变成无限带宽，也就是说，信号经过窗函数处理后，在频域的能量与分布都被扩展了，有效地减少了频谱能量泄露。

加窗口处理，相当于对原始信号进行调制，如下代码所示（原始数据的虚部是 0，不需要处理）：

```
for(int i = 0; i < hfft->count; i++)
{
    TF[i].re = TF[i].re * hfft->win[i];
}
```

不同的窗函数对信号频谱的影响是不一样的。比如最简单的矩形窗，实际上就是对信号不做任何处理，简单地按照时间片段截取一定长度的信号进行处理。本章做频谱计算时选用了汉宁窗

（hanning window），汉宁窗的作用是分析带宽加宽，但是降低了频率分辨率。汉宁窗的数学定义如下：

$$w(t) = (0.5 - 0.5\cos(\frac{2\pi t}{N-1}))R(t) \tag{15-13}$$

其中 $R(t)$是原始信号，t 的范围是 $0 \leqslant t < N-1$，对于其他范围的值，$w(t) = 0$。图 15-5 显示了汉宁窗对信号截取的示意图，以及对频域转换结果的影响。蓝色区域是窗口覆盖的数据部分，白色区域的数据将被削弱。

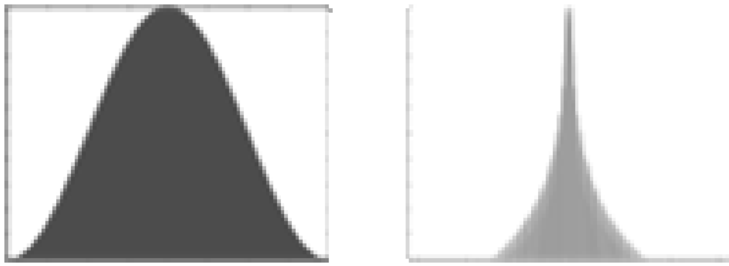


图 15-5 汉宁窗信号截取示意图

使用了窗口，就需要讨论窗口的滑动问题，也就是窗口重叠的处理。用汉宁窗截取的信号片段，可以看出来窗口中部分信号被削弱了（造成衰减），为了抵消部分窗口对信号造成的衰减，各种窗函数都需要对信号进行相应的重叠处理。本节选择的重叠处理方式是选取信号时每次滑动半个窗口位置，使得每个窗口的后半个窗口的衰减在下个窗口的前半个窗口中得到一定的补偿。

2. 计算功率频谱

离散傅里叶变换得到的频域数据是复数，利用一些公式可以根据实部和虚部的值推断出其在时域内的一些特征，比如相位、时延等，不过我们关心的是信号强度，根据频域数据计算相对信号强度的计算公式是：

$$\text{power} = 20.0 \times \log_{10}(\frac{\sqrt{\text{real}^2 + \text{img}^2}}{N/2}) \tag{15-14}$$

计算一段音频数据功率谱的算法非常简单，就是从原始数据中取 N 个采样点的数据，转换到频域，计算出各个频率的信号强度，然后从原始信号偏移 $N/2$ 个采样点位置，再取 N 个采样点的数据进行转换并计算信号强度，与上一次计算的值累加，重复上述过程，直到原始信号处理完毕。具体的算法实现如下代码所示：

```
bool PowerSpectrum(FFT_HANDLE *hfft, short *sampleData, int totalSamples, int channels, float *power)
{
    int i,j;
```

```
    for(i = 0; i < hfft->count; i++)
        power[i] = (float)0.0;

    COMPLEX *inData = new COMPLEX[hfft->count];
    if(inData == NULL)
        return false;

    int procSamples = 0;
    short *procData = sampleData;
    while((totalSamples - procSamples) >= hfft->count)
    {
        procData = sampleData + procSamples * channels;
        for(j = 0; j < hfft->count; j++)
        {
            SampleDataToComplex(procData, channels, &inData[j]);
            procData += channels;
        }
        procSamples += (hfft->count / 2); /*每次向后移动半个窗口*/

        FftWindowFunction(hfft, inData);
        FFT(hfft, inData);

        for(i = 0; i < hfft->count; i++)
        {
            power[i] += float(20.0 * log10(sqrt(inData[i].re * inData[i].re + inData[i].im *
                inData[i].im) / (hfft->count / 2)));
        }
    }

    delete[] inData;

    return true;
}
```

15.3.3 音频播放时实时频谱显示的例子

采样率为 T 的音频数据经过 N 点 FFT 转换后，得到 $N/2$ 个有效的频率和功率分布（另外 $N/2$ 个点的数据具有对称性）。FFT 算法选择的 N 通常都比较大（一般都大于 512），全部显示这么多点的频谱既不现实，也浪费资源。一般频谱最多显示 32 个波段（我用的 Winamp 2.91 版本只有 19 个频谱波段），这就涉及另一个问题，那就是如何从这么多频率数据中选择 32 个用作频谱的显示。

1. 频率范围选择和波段设置

假如我们有 1024 个有效的频率数据，如何选择 32 个数据组成 32 段频谱显示？选取的原则是要选择有代表性的频率，两个波段的中心频率最好不要相差太小，可以是均匀选择，也可以是不均匀选择。可采用的方法很多，最简单的方法就是每隔 32 个点选择一个数据，刚好选择 32 个点的功率值，然后映射到 32 个频谱上显示。还有一个方法是将 1024 个点分成 32 段，每段 32 个点，分别计算每一段的 32 个点的功率平均值，然后将 32 段的功率平均值映射到 32 个频谱上显

示。本章随后给出的实例程序采用的方法是将 1024 个点分成 32 段，然后找到每段的中间点，从中间点向左和向后各取两个点的值，共 5 个值作为采样数据计算的依据，然后赋予不同的权重，中间点权重最高，向两边依次降低，然后计算 5 个点的加权平均值，将加权平均值映射到频谱上显示。

```
void UpdateSpectrum(short *sampleData, int totalSamples, int channels)
{
    float power[FFT_SIZE];
    if(PowerSpectrum(&m_hFFT, sampleData, totalSamples, channels, power))
    {
        int fpFen = FFT_SIZE / 2 / BAND_COUNT;

        int level[BAND_COUNT];
        for(int i = 0; i < BAND_COUNT; i++)
        {
            int centPos = i * fpFen + fpFen / 2;
            double bandTotal = power[centPos - 2] * 0.1 + power[centPos - 1] * 0.15 + power[centPos]
                * 0.5 + power[centPos + 1] * 0.15 + power[centPos + 2] * 0.1;
            level[i] = (int)(bandTotal + 0.5);
        }
        m_SpectrumWnd.SetBandLevel(level, BAND_COUNT);
    }
}
```

15

2. 听觉与视觉延时

由于声音和视觉信号在人类的神经和大脑之间传导过程存在差异，会导致声音和视觉在大脑中的反应有一个时间差，再加上声和光的传播速度本身也有很大的差异，因此，为了使频谱显示能有更好的感官体验，需要对频谱显示的时机做一些调整。一般来说，应该先将声音播放出来后再显示频谱，这就涉及一个问题，即声音的分段多长比较合适。这实际上是播放器音频缓冲区大小的选择问题，缓冲区不能太大，比如 0.5 秒以上的音频缓冲区，等播放完 0.5 秒后再显示频谱，视觉体验上就觉得对不上，鼓声都响了半天了频谱上才体现出来，这种感觉肯定不好。缓冲区太小也不好，首先离散傅里叶转换计算量大，需要一定的时间对音频数据进行处理，缓冲区太小的话就没有足够的时间进行计算，当然，现在的 CPU 都很强劲，这个不是主要问题，主要问题是如果缓冲器太小会导致频谱刷新太频繁，这使得频谱显示看起来不连贯，很机械。这方面我也没有理论的数据支撑，根据实践经验，音频缓冲区大小在 0.05 秒到 0.2 秒之间时，可以取得比较好的视觉体验，本节给出的例子程序使用了 0.1s 的音频缓冲区，对于我的感觉来说，效果还可以。

3. 设计频谱显示器

频谱显示窗口的设计没什么技术难度，只要熟悉 Windows GDI 编程，实现一个频谱窗口应该没有问题。每一个波段的显示内容包含三个部分，如图 15-6 所示，分别是背景、当前强度级别和一条缓缓落下的细线（Top_Bar）。除了需要一个列表记录当前各个波段的强度级别之外，还需要一个列表记录各个波段的 Top_Bar 的位置，每当一个 buffer 播放完成以后，`UpdateSpectrum()`

函数会计算出相应波段的强度，并刷新当前各个波段的强度级别列表，根据选择的播放缓冲区 buffer 大小，刷新的频率应该在每秒 5 ~ 10 次左右。与此同时，内部的位置更新定时器也在周期地减少各个波段的强度级别的值，并降低 Top_Bar 的位置，为了使频谱显示平滑一点，更新定时器的频率要大于强度级别的刷新频率，一般应该在每秒 15 次以上。

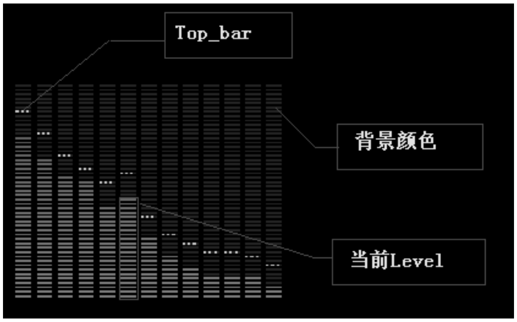


图 15-6　频谱显示的主要元素

Top_Bar 位置和强度级别的刷新就是一个不断减少的过程，但是减少的方式不一样。强度级别的减少可以是一个固定值，每次都减少一定的数量。Top_Bar 则需要维持一个悬停时间，在悬停时间内位置不变化，悬停时间结束后，其值的减少是一个逐步加快的过程，并最终在强度级别减到 0 之前赶上强度级别的位置，这样使得频谱显示看起来生动有趣。

频谱显示窗口是一个需要高速绘图的窗口，直接使用 GDI 函数画频谱窗口已经被证明是低效的方法，不推荐使用。一般都是采用位图缓冲区的方式处理高速刷新的窗口，具体做法就是在一片位图数据中直接通过颜色值控制"生成"频谱显示的位图，然后用贴图的 GDI 函数直接"贴"到窗口 DC 上。

本书在撰写过程中创建的例子程序是一个 Wave 文件播放程序，播放并显示一个跳动的频谱，外观仿 Winamp 的显示效果，绘制出来的频谱形状比较接近 Winamp 的显示，图 15-7 是演示程序最终的效果，所有的代码都包含在本章附带的例子工程中，读者可自行研究。

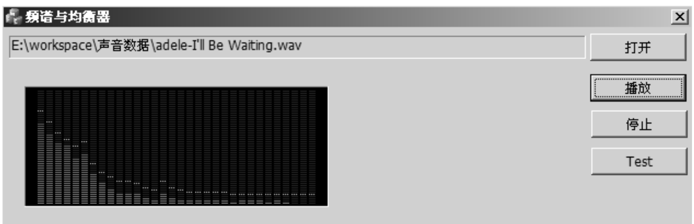


图 15-7　频谱显示示例程序

15.4 破解电话号码的小把戏

2012 年 9 月的时候，一个南京的大学生从电视台播放的一段记者采访 360 总裁周鸿祎的视频中破解了周鸿祎的手机号码，一时被网络热炒。后来，又听说某人买车的时候使用电话银行付款，结果被人录下声音，破解了银行卡号和密码，导致存款被盗。故事情节就像好莱坞电影一般充满技术噱头，但是如果说穿了技术原理，恐怕真的没有什么技术含量。本节就来揭示一下这种小把戏的技术含量，我的目的不是教你破解别人的电话号码做坏事，不过看完本节，你应该知道，当着一群不怀好意的人（他们手里可能拿着手机、录音笔等各种录音器具）之面使用电话银行，真的是非常不明智的行为。

15.4.1 拨号音的频谱分析

首先要说一下，根据拨号音破解电话号码只适用于使用“双音多频技术”（DTMF）的电话设备，老式的拨号盘电话（脉冲式电话机）不适用（估计你也没有这玩意了）。前面介绍过，一些在时域内并不明显的信号特征转到频域以后，其相应的特征便一目了然。对拨号音的分析，也是循着这个思路，首先来看看双音频电话拨号音的频域特征。

1. 双音频电话拨号音

双音多频技术是贝尔实验室的发明，就是将电话机的拨号键盘分成 4×4 的矩阵，每一行对应一个低频信号，每一列对应一个高频信号，如图 15-8 所示：

低频组/Hz	高频组/Hz			
	1209	1336	1447	1633
697	1	2	3	A
770	4	5	6	B
852	7	8	9	C
941	*	0	#	D

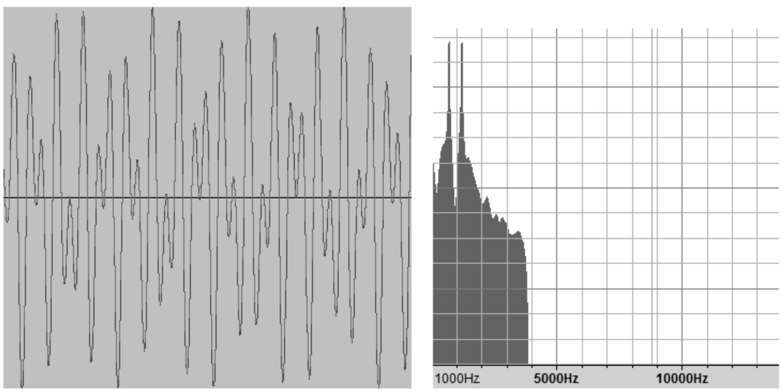
图 15-8 电话键盘双音频对照表

其中低频信号和高频信号的频率都在人耳可以识别的频率范围之内。打电话拨号的时候，每按下一个键，就产生一个高频信号和低频信号的正弦信号组合，局端的电话交换机从这个组合信号中解出两个频率，就知道是那个按键被按下了。

2. 双音频电话拨号音频谱的规律

既然每个拨号音都是由一个高频和一个低频的正弦信号组合，那么它们的频域必然含有两个能量峰值，而且这两个能量峰值分别位于这一高一低两个频率点上。这就是双音频电话拨号音的频谱规律，不同按键对应的拨号音的区别仅仅就是高、低两个频率点不同而已。下面以按键“1”

的音频为例，看看其时域和频域的特征对比，图 15-9 就是按键“1”的音频在时域和频域的形态。



(a)　　　　(b)

图 15-9　按键“1”对应的音频在时域和频域的形态

从图 15-9b 可以看出，虽然受录音杂波影响很大，但是还是可以很明显看到在 697Hz 和 1209Hz 位置上，相对能量强度达到了最大值。

15.4.2　根据频谱数据反推电话号码

了解了电话拨号音的频谱特征，现在就来分析一下如何根据频谱特征反推电话号码。如果采用 *N* 点 FFT 算法，`PowerSpectrum()`函数可以计算出 *N*/2 个有效的频谱数据（还记得对称性吗？），这是破解电话号码的第一步。对于拨号音来说，这 *N*/2 数据中有两个极大值，分别位于这个拨号音对应的高音频率点和低音频率点，所以，破解电话号码的第二步就是找出这两个极大值点。从 *N*/2 个数值中找出最大的 2 个，这是个典型的 top_*n* 算法，可以从互联网上找到很多这样的例子代码，大同小异的点主要就是对长度为 *n* 的有序组的维护方法，目前普遍认为用堆是最高效的方法。不过对于本例的需求，*n*=2 的有序组维护起来其实很简单，交换两个值就可以保证有序表有序。这里给出一个 top_2 算法实现：

```
void ExchangeIndex(int *index, float *power)
{
    if(power[index[1]] > power[index[0]])
    {
        int t = index[0];
        index[0] = index[1];
        index[1] = t;
    }
}

void SearchMax2FreqIndex(float *power, int count, int& first, int& second)
{
    int max2Idx[2] = { 0, 1 };
```

```
    ExchangeIndex(max2Idx, power);
    for(int i = 2; i < count; i++)
    {
        if(power[i] > power[max2Idx[1]])
        {
            max2Idx[1] = i;
            ExchangeIndex(max2Idx, power);
        }
    }

    first = min(max2Idx[0], max2Idx[1]);
    second = max(max2Idx[0], max2Idx[1]);
}
```

计算结果中 `first` 返回低频点的位置，`second` 返回高频点的位置。

破解电话号码的第三步就是根据两个极大值点的位置，反算出对应的频率。根据 15.3.1 节的分析，转换后的结果和频率存在如下对应关系：

$$\text{Freq} = (n-1)T/N \quad （n\text{是 power 数组的位置，}T\text{是采样率）} \tag{15-15}$$

由于频域分辨率的关系，反算出来的频率不会刚好就是图 15-8 中的值，但是应该是非常接近这些值的，可以通过简单的查表定位到真实的频率值，有了这两个真实的频率值，也就知道电话号码了。

现在明白了吧，只要有一套音频分析软件，就可以对拨号音进行分析，并破解电话号码。即使是编程实现，也没有太大的技术难度。看来，以后打电话的时候，还是把拨号音关掉吧。

15.5 离散傅里叶逆变换

有从时域转换到频域的方法，就必然有从频域转换到时域的方法，相对于离散傅里叶变换，这个反向转换就是离散傅里叶逆变换（IDFT）。和离散傅里叶变换一样，离散傅里叶逆变换也是连续傅里叶逆变换的离散形式，先来看看非周期信号连续傅里叶逆变换的公式：

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)\mathrm{e}^{\mathrm{i}\omega t}\mathrm{d}\omega \tag{15-16}$$

连续傅里叶逆变换中的函数 $X(\omega)$是频域连续的，现在假设在 $X(\omega)$的某一段连续区间上按照频域抽取 N 个频率，得到 N 个采样点，则每个采样点的离散傅里叶逆变换公式就是：

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)\mathrm{e}^{\mathrm{i}\frac{2\pi}{N}kn} \qquad n = 0,1,\cdots,N-1 \tag{15-17}$$

如果引入常量 W_N，式(15-17)可以简单记为：

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)W_N^{-nk} \qquad n = 0,1,\cdots,N-1 \tag{15-18}$$

15.5.1　快速傅里叶逆变换的推导

对应于前面介绍的快速傅里叶变换，也存在与之对应的快速傅里叶逆变换（Inverse Fast Fourier Transform，IFFT）。和快速发傅里叶变换算法的推导过程一样，快速傅里叶逆变换算法的推导也是从式(15-18)开始，利用 W_N 的周期性和对称性，将离散傅里叶逆变换逐级分解，减少计算量。具体的推导过程与快速傅里叶变换类似，读者可参考 15.2.2 第 2 小节的过程自行推导，此处不再赘述。

就 IFFT 算法的实现而言，其过程和 FFT 算法的实现一样，只需对 FFT 算法稍作修改，就成为了 IFFT 算法。对比式(15-18)和式(15-3)，可以看出，二者的区别主要有两点：一个是蝶形变换的旋转因子不同，另一个是 IFFT 算法需要对整体结果除以 N。FFT 算法的蝶形变换旋转因子是 W_N^n，而 IFFT 算法的旋转因子是 W_N^{-n}，除此之外，二者蝶形变换的距离和位置关系都是一样的，也就是说，最终位序重排的方法也一样。

15.5.2　快速傅里叶逆变换的算法实现

快速傅里叶逆变换算法的蝶形变换旋转因子是 W_N^{-n}，由式(15-12)可知，其分解的复数形式中余弦项（实部）与 FFT 算法的余弦项相同，正弦项（虚部）的符号位与 FFT 算法的正弦项刚好相反，因此，算法实现仍然可以可用 `FFT_HANDLE` 中的正弦项和余弦项表。IFFT 的算法实现如下：

```
void IFFT(FFT_HANDLE *hfft, COMPLEX * FD2TD)
{
    int i,j,k,butterfly,p;

    int power = NumberOfBits(hfft->count);

    for(k = 0; k < hfft->count; k++)
        FD2TD[k] = FD2TD[k] / COMPLEX(hfft->count, 0.0);

    /*蝶形运算*/
    for(k = 0; k < power; k++)
    {
        for(j = 0; j < 1<<k; j++)
        {
            butterfly = 1 << (power-k);
            p = j * butterfly;
            int s = p + butterfly / 2;
            for(i = 0; i < butterfly/2; i++)
            {
                COMPLEX t = FD2TD[i + p] + FD2TD[i + s];
                FD2TD[i + s] = (FD2TD[i + p] - FD2TD[i + s]) * COMPLEX(hfft->wt[i*(1<<k)].re,
                    -hfft->wt[i*(1<<k)].im);
                FD2TD[i + p] = t;
            }
        }
    }
    /*----按照倒位序重新排列变换后信号----*/
```

```
        for (k = 0; k < hfft->count; k++)
        {
            int r = BitReverise(k, power);
            if (r > k)
            {
                COMPLEX t = FD2TD[k];
                FD2TD[k] = FD2TD[r];
                FD2TD[r] = t;
            }
        }
    }
```


15.6 利用傅里叶变换实现频域均衡器

调节均衡器改变声音的回放效果，就像在汤里放味精一样，掩盖了音乐原始的味道，也能获得一些意想不到的效果。但是，同学，你关注过它的实现原理吗？这一节，我们就来研究一下均衡器的实现原理，同时结合前面介绍的快速傅里叶变换和快速傅里叶逆变换，实现一个可以对各种频率的声音进行精准控制的频域均衡器算法。

从应用角度理解，音乐均衡器有两种常见类型，一种是图示均衡器（Graphic Equalizer），另一种是参量均衡器（Parametric Equalizer）。图示均衡器是一种按照一定的规律把全音频 20 ~ 20000 Hz 划分为若干的频段，每个频段对应一个可以对电平进行增益或衰减的调节器，可以根据需要，对输入的音频信号按照特定的频段进行单独的增益或衰减。参量均衡器不划分固定的波段，可对任意一个频率点（包括频点附近指定频率带宽内的所有点）进行控制，通过调整带宽，使得调节控制可精确（小带宽），也可模糊（大带宽），非常灵活。参量均衡器操作控制不直观，多用在对声音精确控制的专业场合。像 Winamp 和 Foobar 这样的音频播放器，多采用图示均衡器，通过一个带调节器的图形面板可以让用户很方便地对特定频段进行调节。

从信号形态角度理解，均衡器又可以分为时域均衡器和频域均衡器两种类型。时域均衡器对时域音频信号通过叠加一系列滤波器实现对音色的改变，无论是传统的音响设备还是众多音乐播放软件，绝大多数都是使用时域均衡器。时域均衡器通常由一系列二次 IIR 滤波器或 FIR 滤波器串联组合而成，每个波段对应一个滤波器，各个滤波器可以单独调节，串联在一起形成最终的效果。但是，传统的 IIR 滤波器具有反馈回路，会出现相位偏差，而 FIR 滤波器会造成比较大的时间延迟。另外，如果使用 IIR 或者 FIR 滤波器，均衡器波段越多，需要串联的滤波器的个数也越多，运算量也越大。频域均衡器是在频域内直接对指定频率的音频信号进行增益或衰减，从而达到改变音色的目的。频域均衡器没有相位误差和时间延迟，而且不固定波段，可以对任意频率进行调节，不仅适用于图示均衡器，也适用于参量均衡器。特别是采用快速傅里叶变换这样的算法，可以进行更快速的运算，即便是多段均衡器也不会引起运算量的增加。

15.6.1 频域均衡器的实现原理

总体上说，频域均衡器的实现原理很简单，就是将时域音频信号转换到频域，然后对特定频

率进行增益或衰减计算，最后再将结果转换到时域，从而实现对音频音色的修改。如果是多个音轨的音频，需要对每个音轨单独做上述转换和调节。原理简单，但是实现起来并不简单，有很多细节问题需要解决。首先，用户在图示均衡器上拉动拉杆，调节了某个波段之后，这个调节的相对变化如何转化为对频域信号的处理？

15.6.2 频域信号的增益与衰减

图示均衡器允许用户调节每个波段的增益和衰减，调节的单位通常是 dB（分贝）。dB 是一个相对比值，用于表示两个值之间的比例关系。20 dB 的信号的实际强度是 0 dB 信号的 10 倍，而 –20 dB 的信号的实际强度是 0 dB 信号的 1/10。当用户调节了某个波段的增益值后，如何将这个相对增量转换成能在频域内直接对频域数据进行计算处理的增益强度，是频域均衡器需要解决的重点问题。

1. 频域的增益和衰减

首先，增益或衰减是基于频率（频段）进行计算，所以这个问题需要在频域内处理。处理的方法就是式(15-14)所描述的功率相对强度与频域信号值的计算关系。与处理频谱的方式不同，这里要通过这个公式反推需要在频域信号叠加什么值才能使得功率达到指定的增益或衰减。具体来说，就是频域信号的实部和虚部各需要叠加什么值，以及这种叠加关系是什么。

为了给某个频率的信号增益 P_z 个 dB（P_z 若是小于 0，表示是对信号进行衰减），根据公式(15-14)，需要叠加的量是$(x+y_i)$。现在对 x 和 y 赋予不同的权重，不妨设实部的权重是 0.75，虚部的权重是 0.25，也就是令 $x=0.75k$，$y=0.25k$。将 x 和 y 代入到式(15-14)，简化后可以得到：

$$P_z = 20.0\times\log_{10}(\frac{\sqrt{10}}{2N}k)$$

对于指定的增益（或衰减）值 P_z，可以利用上式计算出 k 的值。接下来，假设某个频率在频域的值是$(a+b_i)$，其相对强度是 P_0，如果给其叠加一个增益（或衰减）P_z，需要的计算是：

$$P_0 + P_z = 20.0\times\log_{10}(\frac{\sqrt{a^2+b^2}}{N/2}) + 20.0\times\log_{10}(\frac{\sqrt{10}}{2N}k)$$

$$= 20.0\times\log_{10}(\frac{\sqrt{(\frac{\sqrt{10}}{2N}ka)^2+(\frac{\sqrt{10}}{2N}kb)^2}}{N/2})$$

由上式可知，需要给原始信号进行叠加的值是 $\sqrt{10}k/2N$，叠加的方式是复数乘法。

前面给出的快速傅里叶变换和逆变换都是复数变换，但是处理音频数据时都只使用了复数的实部（虚部赋值为 0.0），因此，叠加值在频域计算好之后，需要转换到时域，将虚部清 0，只保留实部的值，然后再转换到频域，此时的叠加值才是最终参与复数乘法计算的值。根据增益值计

算叠加量的算法实现如下：

```
bool UpdateFilter(EQUALIZER_HANDLE *hEQ, float *gain, int count)
{
    if((hEQ->hfft.count / 2) < count)
        return false;

    for(int i = 0; i < hEQ->hfft.count / 2; i++)
    {
        double dbk = pow(10.0, gain[i]/20.0);
        hEQ->filter[i].re = (float)(dbk * 0.75);
        hEQ->filter[i].im = (float)(dbk * 0.25);
        hEQ->filter[hEQ->hfft.count - 1 - i].re = hEQ->filter[i].re;
        hEQ->filter[hEQ->hfft.count - 1 - i].im = hEQ->filter[i].im;
    }

    IFFT(&hEQ->hfft, hEQ->filter); //to time-domain
    for(int i = 0; i < hEQ->hfft.count; i++)
    {
        hEQ->filter[i].im = (float)0.0;
    }
    FFT(&hEQ->hfft, hEQ->filter); //to freq-domain

    return true;
}
```

算法中只计算了前一半的叠加量,后一半采用的是对称赋值,这是由频域信号的对称性决定的。

2. 应用三次样条曲线插值算法平滑增益与衰减

对均衡器调节，对应的是一个波段，不是一个频率。因此，在频域进行增益（或衰减）计算时，不应仅考虑一个频率，而应考虑以这个频率为中心的整个波段。当然，也不是整个波段都进行相同的增益（或衰减），最好的方法是波段的中心频率点执行最大增益（或衰减），然后按照波段带宽，从中心到边缘逐步降低增益（或衰减）的值。

从波段中心到边缘的变化可以采用线性方式，从示意图看起来就是多条折线。当然，也可以采用当前流行的方法，就是采用曲线插值的方法，使示意图看起来像一条平滑的曲线。说到曲线插值，小伙伴们应该想到第 12 章介绍的三次样条曲线拟合算法。是的，本章的均衡器例子就使用三次样条曲线插值算法，得到一条平滑的增益（或衰减）值曲线。生活中到处都是算法，不是吗？代码正如 UpdateEqCurve()函数所示，InterpolationX 和 InterpolationY 是插值点的增益（或衰减）值，对应的是所有波段的中心点频率和两个附加的起始点和终点，使用三次样条曲线拟合的算法得到整条曲线的值。

```
void UpdateEqCurve()
{
    float gain[FFT_SIZE/2];
    SplineFitting eq;

    eq.CalcSpline(InterpolationX, InterpolationY, EQ_BAND_COUNT + 2, 1, 0.0, 0.0);
    for(int x = 0; x < FFT_SIZE/2; x++)
```

```
        {
            gain[x] = (float)eq.GetValue(x);
        }
        UpdateFilter(&m_hEQ, gain, FFT_SIZE/2);
    }
```

15.6.3 均衡器的实现——仿 Foobar 的 18 段均衡器

有了以上的分析，均衡器算法的实现就水到渠成了，将音频数据按照 FFT 算法一次能处理的最大数据分块，对每一块音频数据用 FFT 算法将其转换到频域，对信号进行计算，然后在用 IFFT 算法将音频数据转换到时域，每一块的处理算法如下：

```
FFT(&hEQ->hfft, leftData);
  if(channels > 1)
  {
        FFT(&hEQ->hfft, rightData);
  }

  SampleDataMpGain(leftData, rightData, hEQ->hfft.count, channels, hEQ->filter);
  IFFT(&hEQ->hfft, leftData);
  if(channels > 1)
  {
        IFFT(&hEQ->hfft, rightData);
  }
```

SampleDataMpGain()函数负责增益（或衰减）的计算，就是将信号与 filter 逐个做复数乘法运算，最终的结果将在逆变换后得到的音频数据中得到体现。

最后需要注意的是，对信号进行增益（或衰减）计算可能会导致信号超出合法值的范围，从听觉上理解就是会导致调整后的声音听起来有杂音，因此需要在转换过程中消除这种现象。在音频处理领域，有很多专门的算法应对这种情况，很多个人和组织都申请了很多这样的专利。如果要实现一个专业的均衡器，你需要研究这些算法，本章的例子只是为了演示用 FFT 算法实现频域均衡器的原理，所以采用了一种简单的处理方法，就是当有信号值越界后，简单调整成最大值。这个调整在 ComplexToSampleData()函数中实现，这个函数与 SampleDataToComplex()函数对应，用于将调整后的信号转换成 PCM 格式的音频数据。

```
void ComplexToSampleData(COMPLEX *cdl, COMPLEX *cdr, int channels, short *sampleData)
{
    if(cdl->re > 1.0)
        cdl->re = 1.0;
    if(cdl->re < -1.0)
        cdl->re = -1.0;

    *sampleData = short(cdl->re * 32768.0);
    if(channels != 1)
    {
        if(cdr->re > 1.0)
            cdr->re = 1.0;
        if(cdr->re < -1.0)
```

```
            cdr->re = -1.0;
        *(sampleData + 1) = short(cdr->re * 32768.0);
    }
}
```

至此，所有的核心算法都已经完成，按照惯例，可以做个例子演示一下了。是的，来一个仿 Foobar 的 18 段均衡器吧，顺带体现一下三次样条曲线插值算法的价值。图 15-10 就是演示程序，均衡器曲线是我随便调的，没人会这么调均衡器吧？完整的示例程序代码包含在本章的随书代码中，除了算法核心代码之外，剩下的都是常规的 Windows 编程，请大家自己研究吧。

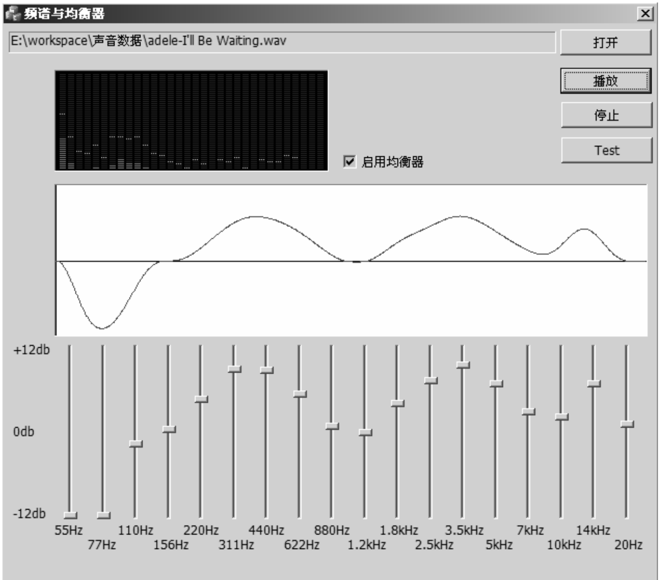


图 15-10　一个仿 Foobar 的 18 段均衡器的例子

15.7　总结

本章介绍了离散傅里叶变换及其快速算法（FFT）的几个应用例子，都是生活中常见的功能，背后隐藏的却是如此简单的算法实现。其实离散傅里叶变换在工业和信号处理领域有非常广泛的应用，并不仅限于本章的例子。本章给出的算法不算最高效的算法实现，但是中规中矩，是研究算法原理的好例子，读者还可以从互联网上找到处理实数的更高效的 FFT 算法来研究。我的目的是让大家再次了解生活中隐藏的算法，解除对算法的神秘感，不知道是否达到了？

15.8　参考资料

[1] E.O.布里汉. 快速傅里叶变换. 柳群译. 上海：上海科学技术出版社，1979

[2] 何振亚. 数字信号处理的理论与应用. 北京：人民邮电出版社，1983

第 16 章 全局最优解与遗传算法

最优化问题永远是算法中永恒的话题之一，之前我们介绍过一些求最优解的常用算法，比如不一定能得到最优解的贪心算法，适用于多目标、多阶段优化的动态规划算法，还有理论上万能、但是实际应用中常常受问题规模限制的穷举算法。这些算法都是采用一种确定或几乎确定的方式寻找最优解，本章我们将介绍一种完全不同的方式寻找最优解，那就是充满随机性的启发式方法。这里提到的启发式方法和之前介绍搜索算法时常常提到的启发式搜索是两个概念，大家不要搞混了。

传统的最优解算法都是建立在确定性基础上的搜索，在搜索过程中遇到一个决策点时，对于选 a 还是选 b，其结果是确定的。比如贪婪法，就是按照贪婪策略选择，同样的条件下，每个决策选 1000 次结果都是一样的。随机化算法就不会有这么确定的结果，它是一种带启发式的随机搜索，但是随机化算法并不是闭着眼睛掷骰子，各种随机化算法都有与之对应的理论基础。此类随机化算法常见的有模拟退火算法、禁忌搜索、蚁群算法、神经网络，当然也包括本章要介绍的遗传算法（genetic algorithm）。这些模拟、演化（进化）式的启发式搜索算法的搜索过程不依赖目标函数的信息，非常适合一些传统最优化方法难以解决的复杂问题或非线性问题，在人工智能、自适应控制、机器学习等领域得到了广泛的应用。

16.1 遗传算法的原理

达尔文（Darwin）的进化论讲述的是“物竞天择，适者生存”的自然原理，生物体通过自然选择、基因突变和遗传等规律进化出适应环境变化的优良品种。遗传算法就是这样一种借鉴生物体自然选择和自然遗传机制的随机搜索算法，其搜索过程就是“种群”一代一代“进化”的过程，通过评估函数进行优胜劣汰的选择，通过交叉和变异来模拟生物的进化。优胜劣汰是这种搜索算法的核心，根据优胜劣汰的策略不同，算法最终的效果也各不相同。

遗传算法将问题的解定义为进化对象的个体，对若干个体组成的种群进行选择、交叉（杂交）和变异处理，每处理一次种群就“进化”一代。只要评估和选择策略合适，经过若干次“进化”

之后，种群中就会出现比较接近最优解的个体，对应的就是问题的近似优化解。这就是遗传算法的原理，接下来我们要详细介绍遗传算法的处理流程，在这之前，先来了解几个与遗传算法有关的基本概念。

16.1.1 遗传算法的基本概念

遗传算法是借鉴生物进化过程而提出的一种启发式搜索算法，因此在介绍遗传算法之前，首先要普及几个基本的生物学术语。

首先是基因（gene）和染色体（chromosome），很多介绍遗传算法的资料通常将这二者混为一谈，其实从生物学角度看，这是两个不同的概念。基因是一个单独的遗传因子，包含一组不能再拆分的生物学特征，而染色体则可以理解为是一组基因的组合，如无特殊说明，本书用“基因”一词表示参与计算的遗传特征。

接下来是种群（population）和个体（individuals），生物的进化以群体的形式进行，这样的一个群体就称为种群，种群中的每个生物体就是一个个体。种群中的每个个体是相互联系，相互影响的，这种联系影响着种群的进化。

接下来是残酷的“适者生存”，只有对环境适应度高的个体，生存能力强，繁衍的后代会比较多。相反，环境适应度低的个体参与繁殖的机会比较少，后代会越来越少，最后只剩下强者。

最后是遗传和变异，下一代个体会遗传上一代个体的部分基因，使得个体的生物学特征能够延续到下一代。但是遗传并不是平稳的，会有一定的概率发生基因突变，基因突变所产生的新的生物学特征可能会提高个体的环境适应度，也可能会降低个体的环境适应度。能提高个体的环境适应度的突变基因通过适者生存原则被种群延续到下一代。

生物通过繁殖产生下一代，在遗传算法看来，繁殖就是基因交叉（crossover）算法的处理过程，将种群中的个体两两进行部分基因编码片段的互换，即可得到下一代的个体。遗传算法中的基因突变（mutation）算法是通过直接替换个体基因中的某个或某几个编码实现的，也有的算法采用直接生成一个新的个体（相当于替换全部基因编码）来实现基因突变。基因交叉和基因突变都是遗传算法的重要步骤，但是不能进行得太频繁，如果进行得太频繁会导致每一代的基因差异太大，最终使得算法无法收敛到近似最优解。基因交叉和基因突变如果发生得太少也不行，因为这样无法保证种群的多样性，最终使得算法可能收敛到某个局部最优解，从而无法得到全局最优解。一般遗传算法的实现都会定义一个基因交叉发生概率和基因突变发生概率，通过这两个概率控制其发生的频度。

选择（selection）也是遗传算法中的重要算法之一，选择就是根据个体的适应度，按照一定的规则从上一代种群中选择一些优良的个体遗传到下一代种群中。适应度（fitness）是个体对环境的适应程度，适应度低的个体会被逐步淘汰，适应度高的个体会越来越多，遗传算法一般都会根据问题要求设置一个适应度函数评估每个个体的适应度。

16.1.2　遗传算法的处理流程

遗传算法的处理流程就是一种模拟计算的过程，如图 16-1 所示。

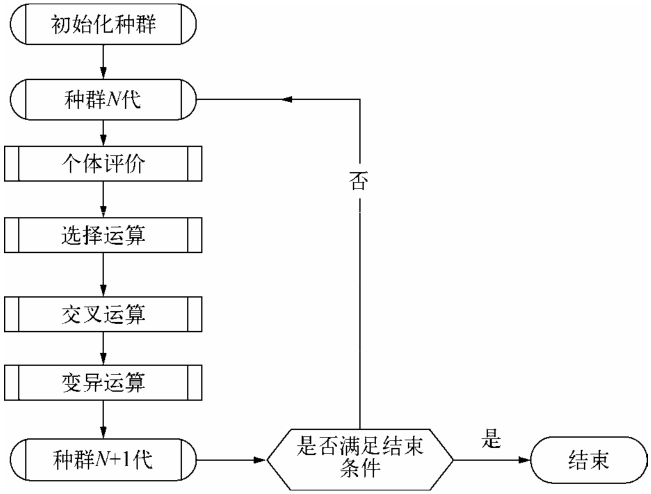


图 16-1　遗传算法模拟计算流程图

种群从第 N 代到第 N+1 代演化的过程中，个体评价、选择运算、交叉运算和变异运算分别扮演着自然界生物进化过程中的“优胜劣汰”“交配繁殖”和“基因突变”所对应的角色，其中选择运算、交叉运算和变异运算被称为遗传算法的遗传算子。

严格来说，遗传算法并不是一个具体的算法，它代表的是一种思想。针对不同问题，基因的选择与编码、适应度评估函数的设计以及遗传算子的设计都是各不相同的。遗传算法其实是一种非常简单的算法，如果你不相信，请看图 16-1 的流程图，全是直线箭头。但是你还是不相信，对吧？虽然原理很简单，但是遇到问题，仍然无从下手。原因在于有几个问题还是没有明确，首先，基因是什么，怎么选择，怎么编码？其次，适应度评估函数如何设计？最后，三个遗传算子如何设计？下面就从这三个方面介绍一下如何设计针对具体问题的遗传算法。

1. 基因的选择与编码

简单地理解，遗传算法中的基因就是以某种编码的形式表示的实际问题的解。确实很简单，举个例子，假如要求解抛物线 $y=-2x^2+x+15$ 在[−2.5, 3.0]区间上的最大值，抛物线函数的自变量 x 是问题空间的解，对应遗传算法中的基因就是 x（$-2.5 \leqslant x \leqslant 3.0$）的某种编码形式。再比如本书第 3 章提到的 0-1 背包问题，包中所选择的物品就是问题空间的解，比如[0,1,0,1,1,0,0]，对应遗传算法的基因就是用某种编码形式存储的这个[0,1,0,1,1,0,0]状态。由此可见，针对不同的问题，基因的形式千变万化，也就不难理解为什么不存在一劳永逸的全能遗传算法了。

说到基因，就不能不提基因的编码，基因可能有点抽象，但是编码是具体的。所谓编码，就

是用计算机能存储和处理的数据形式表达基因所代表的问题的解。遗传算法首先要解决的问题就是基因的选择与编码问题，如果这个问题不解决，遗传算法的三大遗传算子的设计就是空中楼阁。因为遗传算子需要在遗传算法中对基因进行选择、交叉和变异处理，需要选择一种合理的基因编码规则，才能使得遗传算子可以方便快捷地处理基因的选择、交叉和变异计算。基因的编码规则必须满足以下三个条件。

- 完备性：问题空间的所有解在遗传算法中都有编码值与之对应。
- 健全性：遗传算法中的每一个编码值在问题空间中也有对应的值。
- 非冗余性：遗传算法中的编码值与问题空间中的解满足一一对应关系。

遗传算法常用的基因编码方式有二进制编码、格雷编码、符号编码、属性序列编码等方式，对于多参数的最优化问题，可用上述方式对每个参数进行编码，然后用级联或交叉的方式组合成最终的基因编码。二进制编码方式是最简单的编码方式，简单地说，就是直接使用被选择为基因的解，不进行特殊编码，被选为基因的解在内存中是二进制形式存在。比如求抛物线最大值的问题，对 x 不进行任何编码，直接选择若干个在[−2.5, 3.0]区间上的随机数作为初始种群，遗传算子直接对种群中的二进制数据进行基因的交叉和变异计算，评估函数的设计也很简单。二进制编码虽然简单，但是从信息论角度分析，二进制编码存在汉明悬崖（Hamming Cliff）问题[①]，对基因做很小的交叉和变异，得到的结果却差异巨大，会使得遗传算法的基因交叉和变异难以跨越。

我们常用的数字体系，无论是二进制、十进制还是十六进制，每个数位都是有权位的，同样的数字 1，放在个位和放在十位上代表的意义是不同的。格雷码（Gray Encoding）则是一种无权码，其特点是任意两个相邻的格雷码之间只有一位不相同。此外，格雷码的最大数和最小数也仅有一位不同。因此它又称为循环二进制码或反射二进制码，其循环和单步的特性可以消除随机取数时出现重大误差的可能。格雷码最初是作为一种通信领域内的可靠性编码使用，但是其“两个相邻的格雷码之间只有一位不同”的特性可用于遗传算法的基因编码。格雷编码连续性好，可避免汉明悬崖问题，增强遗传算法的局部搜索能力。关于格雷编码和二进制编码之间的转换方法，本章的随书代码中有转换算法，请读者参考本章最后给出的参考资料自行研究转换算法的原理和实现。

对于某些非数字体系的问题，其基因无法直接用数字表示，这就需要用一些符号编码来表示基因。举个学生选课问题的例子，假设有 26 门课程可供学生选择，每个学生可选 4 门课程，很显然，学生所选课程如果作为运算的输入参数，无法用数字表达。但是如果我们定义英文字母 A ~ Z 分别代表 26 门课程，则每个学生所选课程作为输入参数就可以组成符号编码，比如 ACEK，采用符号编码后就可以转换成遗传算法中的基因进行遗传算子计算了。属性序列编码和符号编码

① 在信息论中，汉明距离（Hamming）是描述两条信息之间相似程度的一个属性。两个长度相等的字符串对应位置上的不同字符的个数就是两个字符串的汉明距离。换句话说，如果将一个字符串通过替换字符的方式变换成另一个字符串，需要做替换的次数就是汉明距离。汉明悬崖是二进制编码的一个缺点，就是相邻整数的二进制代码之间有很大的汉明距离，使得遗传算法的交叉和变异变得难以跨越。

的处理思想类似，也是解决非数字体系问题常用的基因编码方式。如果算法的输入参数无法用数字直接编码，但是输入参数由若干个固定个数的属性组成，这些属性变化就可以代表不同的输入参数，在这种情况下，可以为每种属性设置编码，然后按照属性序列排列，得到一个用属性序列编码表示的输入参数，这就是属性序列编码的主要思想。以 0-1 背包问题为例，我们把一次选择完成后背包中的物品作为问题的解，则这个解无法直接用数字进行编码，但是观察这个解，发现其组成就是 7 件物品的选择状态，我们对每个物品的选择状态编码，0 表示不选，1 表示选，最终背包问题的基因编码就是类似[0,1,0,1,1,0,0]这样的形式。

2. 适应度评估函数

在遗传算法中，适应度用于衡量群体中每个个体与最优解的接近程度，也就是个体基因的优良程度。适应度高的个体遗传到下一代的概率比较大，而适应度小的个体遗传到下一代的概率比较小。计算个体适应度的函数就是*适应度评估函数*或*适应度函数*（fitness function），遗传算子中的选择算子需要根据个体的适应度函数来评估每个个体遗传到下一代的概率，因此，适应度评估函数的设计总是和基因的选择紧密相关。

适应度函数对个体的评估过程一般是这样的：首先对种群中个体的基因进行解码处理，从遗传算法中的基因编码转换到问题空间中的数据表达形式；其次，根据问题空间中的数据表达形式，使用问题空间的目标函数或最优值评估方法计算问题空间对应的结果；最后，根据问题的类型和最优解的形式，按照一定的规则对计算的结果进行评估和转换，得到遗传算法中的个体适应度。

遗传算法中对适应度函数的处理有两种方式，一种是在算法的执行过程中始终使用固定的适应度函数，另一种是在算法的不同阶段使用不同的适应度函数。第一种方法的算法处理简单，但是存在运算初期的早熟问题（也称未成熟收敛问题）和运算后期的竞争区分度不高的问题。所谓早熟问题，就是在遗传运算的初期，少数个体的适应度非常高（可能是局部最优解），这样在遗传过程中，这些个体在下一代中所占的比例很高，使得交叉和变异对种群多样性的作用被严重降低，种群多样性无法保证，最终因为局部最优解的存在而错过全局最优解。所谓竞争区分度不高的问题，是在遗传算法运算的后期，此时种群中多数个体的值都已经非常接近最优解，它们之间的适应度非常接近，互相之间的竞争力几乎相同，随机选择时因为适应度几乎一样导致根据概率选择的过程变成等概率的平均选择。一旦出现这种情况，遗传算法的搜索机制实际上就没有重点搜索区域了，变成了随机的平均搜索，即便算法再进行几千代、几万代模拟繁殖，其结果也变化不大，严重影响遗传算法的效率。

第二种方法采用自适应适应度函数或可变适应度函数，在遗传算法的不同阶段使用不同的规则计算个体的适应度，规避使用固定适应度函数可能面临的两个问题。在遗传算法中，这种方式称为*适应度尺度变换*（fitness scaling），常见的适应度尺度变换方式有线性尺度变换、乘方（幂）尺度变换和指数尺度变换。对于简单的问题（或没有局部最优解的情况），使用固定的适应度函数即可，如果要研究可变适应度函数在遗传算法中的应用，读者可以通过本章的参考资料自行研究。

前面已经介绍过，遗传算法是一种随机搜索算法，遗传算法通过适应度函数控制搜索的重点区域，如果适应度函数设计不当，有可能找错重点区域，从而错过最优解。所以，遗传算法的适应度函数是算法是否能成功的关键因素。

3. 遗传算子的设计

选择算子、交叉算子和变异算子被称为遗传算法的三个遗传算子。选择算子又称复制算子，是遗传算法中保证优良基因传播的基本方式，对应的是“适者生存”的群体进化现象。交叉算子对应的是物种“繁殖和交配”产生的基因交换现象，变异算子对应的是“基因突变”这种进化现象，交叉和变异算子用于产生新的个体，是基因多样性的保证。

选择算子的作用就是从群体中选出比较适应环境的个体复制（繁殖）到下一代，选择算子运行的基础是个体的适应度评估值，所以选择算子和适应度函数直接影响着遗传算法的性能。根据“优胜劣汰”的原理，遗传算法的选择算子都是非均匀选择的，常见的选择策略有以下几种。

- 比例选择（proportional selection）：又称“轮盘赌选择”（roulette wheel selection），是一种回放式随机采样方法，每个个体进入下一代的概率等于它的适应度值与整个种群中个体适应度值总和的比例。
- 随机竞争选择（stochastic tournament）：又称“随机锦标赛选择”，每次用比例选择方式从群体中选择两个或多个个体进行适应度竞争，适应度高的个体被选中。重复这个过程，直到下一代个体选满为止。
- 最佳保留选择：确切地说，这是和交叉算子与变异算子结合在一起的一种选择策略。首先用比例选择方式选择下一代个体，但是每次都找出上一代中适应度最高的个体，直接替换到适应度最差的个体，并且这个个体不参与交叉和变异运算，确保它能遗传到下一代。
- 排序选择：对群体中的所有个体按期适应度大小进行排序，根据排序结果，按照某种规则计算出每个个体被选中的概率。
- 确定式采样选择（deterministic sampling）：该策略能确保适应度高的个体 100%被遗传到下一代。具体的方法是：根据个体的适应度计算群体中每个个体在下一代中期望的生存数目，计算方法是 $N_i = M \times F_i / \sum_{i=1}^{M} F_i$ 。用 N_i 的整数部分确定对应的个体在下一代种群中的生存数目，对 N_i 求和得到 $M' = \sum_{i=1}^{M} \lfloor N_i \rfloor$。按照 N_i 的小数部分对个体进行排序，按照从大到小的顺序依次取前 $M - M'$ 个个体加入到下一代种群（每个个体的数量是 N_i），最终得到 M 个下一代种群。

遗传算法中的交叉算子的功能是将两个个体的基因的一部分片段（基因片段对应的位置相同）互相交换，从而产生两个新的个体。设计交叉算子的算法，一般要求既不要太多地破坏个体基因中的优良基因，又要能够有效地产生基因不同的新的个体，保证种群的多样性。交叉算子的设计一般由基因的编码方式决定，基本过程就是随机从群体中选择两个个体配对，然后按照一定的交叉规则交换对应位置上的基因片段。基因交叉规则大致可分为以下几类。

- 单点交叉（one-point crossover）：在配对个体的基因中只随机选择一个点，以随机概率交换这个点对应的基因片段，从而形成两个新个体。
- 两点交叉（two-point crossover）与多点交叉（multi-point crossover）：在配对个体的基因中只随机选择两个或多个点，以随机概率交换每个点对应的基因片段，从而形成两个新个体。
- 均匀交叉（uniform crossover）：又称为一致交叉，对配对个体的基因上的每个点都按照相同的交叉概率交换其对应的基因片段，从而形成两个新个体。
- 算术交叉（arithmetic crossover）：由两个配对个体的线性组合而产生出两个新的个体，该操作对象一般是由浮点数编码表示的基因。

遗传算法中的变异算子的功能是将个体基因上的某个点对应的基因片段替换成适合该点的其他基因片段值，从而产生一个新的基因。和生物学进化的基因突变一样，变异在遗传算法中也只是产生新个体的辅助手段，通常用一个比较低的概率控制变异发生的频度。变异算子和交叉算子共同决定了遗传算法的搜索性能，通过维持种群的多样性避免早熟现象。变异算子主要解决两个问题，一个是如何确定变异的位置，另一个是如何进行基因变异。常用的变异算子类型如下。

- 单点变异（One-point Mutation）：对个体的基因编码随机选择一个点，以随机概率进行变异运算。
- 固定位置变异：对个体基因上的一个或几个固定位置上的基因片段，以随机概率进行变异运算。
- 均匀变异（一致性变异）：对个体基因上的每个片段，都使用均匀分布的随机数，以较小的随机概率进行变异运算。
- 边界变异（Boundary Mutation）：做变异操作时，使用基因编码规则定义的编码边界值（如果有多个边界值，比如同时有最大值和最小值的情况，则根据实现定好的规则选一个或随机选一个）替换原来的基因片段。
- 高斯变异：基因变异的随机概率不是平均分布随机数或普通正态分布随机数，而是采用符合高斯分布的随机数生成器生成随机概率。

具体的变异算法是与基因编码方式有关的，比如二进制编码和浮点数编码，直接将某一位从 1 变成 0，或从 0 变成 1 就实现了变异。对于符号编码的基因，直接将某个位置上的符号替换成符合该位置要求的其他符号即可变成变异。对于属性序列编码方式，改变某个属性的值也算是实现了变异。总之，变异只是一个抽象的要求，具体的算法实现则千姿百态。

4. 遗传算法的运行参数

除了遗传算子和适应度函数，遗传算法的四个重要参数也会影响结果的求解，这四个参数分别是种群大小 M、交叉概率 P_c、变异概率 P_m 和进化代数 T。

- 种群大小 M 表示种群中个体的数量，种群的个体数量决定了遗传算法的多样性。M 值越大，种群的多样性越好，但是会增加算法的计算量，降低运行效率。但是如果 M 值太小，会因为遗传多样性降低而导致比较容易出现早熟现象。一般建议 M 的取值最小为 20。

- 交叉概率 P_c 决定了产生新个体的频度，这是保证种群多样性的关键参数之一。交叉概率太小，会导致新个体产生速度慢，影响种群多样性，但是交叉概率也不能太大，过高的交叉概率会使基因的遗传变得不稳定，优良基因比较容易被破坏。一般建议交叉概率取 0.4 ~ 0.9 之间的值，0.8 是比较常用的值。
- 变异概率 P_m 也是影响群体多样性的参数之一。变异概率太小不利于产生新个体，对种群多样性有影响，但是变异概率太大也会使基因的遗传变得不稳定，优良基因比较容易被破坏。根据遗传学原理，基因突变是一个小概率的事件，在遗传算法中，变异算子对种群的影响也应该远远小于交叉算子。一般建议变异概率取值小于 0.2。
- 进化代数 T 是遗传算法的退出条件，如果 T 太小，会使得遗传算法在种群还没有进化成熟就退出了，自然会影响结果的准确性。当然，T 也不是越大越好，当种群已经接近最优结果的时候，每次进化所产生的变化非常小了，在这种情况下仍然继续进化不仅影响算法的效率，对结果精度的提高也没有太大的帮助。一般建议 T 最小进化 100 代。

16.2 遗传算法求解 0-1 背包问题

本书的第 3 章介绍了 0-1 背包问题，并给出了使用贪婪法求解 0-1 背包问题的算法，除此之外，我们知道这个问题还可以使用动态规划法和穷举法求解。本章介绍了遗传算法，当然这个问题也可以用遗传算法求解。这一节我们将介绍一种使用遗传算法求解 0-1 背包问题的算法实现，该算法采用属性序列方式对基因编码，遗传算子则使用了比例选择模式、多点交叉和均匀变异三种方式实现，虽然核心代码只有 60 多行，但是却实现了基本遗传算法的全部要素。

16.2.1 基因编码和种群初始化

0-1 背包问题基因编码的方式已经在 16.1.2 第 1 小节简单介绍过了，这里我们讨论一下具体的编码实现。基因由 7 件物品的状态组成，1 表示装入背包，0 表示不装入背包，这样一个 7 元组可以用一个数组表示。每个个体除了基因以外，还有适应度、选择概率和积累选择概率。本章的算法给出个体的定义如下：

```
typedef struct GAType
{
    int gene[OBJ_COUNT];
    int fitness;
    double rf;
    double cf;
}GATYPE;
```

种群初始化就是为每个个体选择随机的基因，这一点可以使用一些 0 和 1 的随机数直接填充 gene 属性数组即可。这里需要说明一下，随机填充的基因，也就是物品的装入状态并不一定符合问题要求，比如会出现物品总重量超过背包容量的问题。有两个策略处理这个问题，一种策略是初始化种群基因时判断这种情况，保证每个基因都是满足问题要求的状态。如果采用这种策略，

在交叉算子和变异算子的设计时也要考虑这种情况，当新产生的个体的基因不符合问题要求的时候做特殊处理。另一种处理策略是不判断基因的非法状态，只是在适应度评估时对不符合问题要求的个体给出惩罚性评估值，使其获得一个非常低的选择概率，这样的个体在选择算子的处理过程中自然就被淘汰。第二种策略需要特殊设计适应度函数，给出惩罚机制，在选择算子的设计上也要做特殊考虑，但是总体还说还是比第一种策略简单，所以本章的算法采用第二种策略。

16.2.2 适应度函数

对于 0-1 背包问题，其最优化目标函数就是对背包内装入物品的价值进行评估，取价值最大的那个结果，当然，前提条件是物品的总重量不能超过背包容量。因此，我们把适应度定义为背包内装入物品的总价值，同时对非法状态给出惩罚性的适应度。因为物品的总价值都是比较大的数值（最小值是 10），所以惩罚策略就是将非法状态的个体的适应度评价为 1。当然还可以用更小的值，但是仁慈一点吧，对选择概率来说，这已经足够小了。

```
int EnvaluateFitness(GATYPE *pop)
{
    int totalFitness = 0;
    for(int i = 0; i < POPULATION_SIZE; i++)
    {
        int tw = 0;
        pop[i].fitness = 0;
        for(int j = 0; j < OBJ_COUNT; j++)
        {
            if(pop[i].gene[j] == 1)
            {
                tw += Weight[j];
                pop[i].fitness += Value[j];
            }
        }
        if(tw > CAPACITY) /*惩罚性措施*/
        {
            pop[i].fitness = 1;
        }
        totalFitness += pop[i].fitness;
    }

    return totalFitness;
}
```

EnvaluateFitness()函数是适应度评估函数的实现，totalFitness 变量用于计算种群中全部个体的适应度总和，在这个函数中计算种群的适应度总和是因为选择算子需要计算选择概率。除此之外，这个函数的实现就是上述适应度函数讨论的内容，无需过多说明。

16.2.3 选择算子设计与轮盘赌算法

选择算子的设计采用比例选择方法，也就是轮盘赌选择方法。轮盘赌算法是随机算法中最常用的一种概率选择算法，因原理和赌场中的轮盘赌原理相似而得名。每个个体被选择的概率就像

轮盘上的一个扇区，面积大的扇区被选中的概率就比较大，面积小的扇区被选中的概率就比较小。轮盘赌算法首先需要计算出每个个体的选择概率，这个概率通常用个体的适应度与种群的总体适应度之和的比值来计算。显然，对于适应度大的个体来说，这个比值就比较大，意味着其被选中的概率就比较高，这体现了根据适应度评估优胜劣汰的原则。

传统的轮盘赌算法需要根据种群的适应度总和确定转盘的格式总数，然后根据选择概率确定每个个体对应的格子，最后随机产生一个转动格子的数量，由这个随机数加上当前转轮的起始位置得到最终对应的格子数，从而确定该格子对应的个体被选中。除了直接使用上述方法实现轮盘赌算法，还可以用一种比较简单的方法模拟轮盘赌算法，在介绍这种方法之前，先介绍一下什么是积累概率。

某个个体对应的积累概率定义为该个体的选择概率和前一个个体的积累概率的和，显然这是一个递归定义，如图 16-2 所示，假如某种群中有 8 个个体，每个个体的选择概率分别是 0.1、0.2、0.15、0.25、0.05 和 0.25，则它们的积累概率分别是 0.1、0.3、0.45、0.7、0.75 和 1。

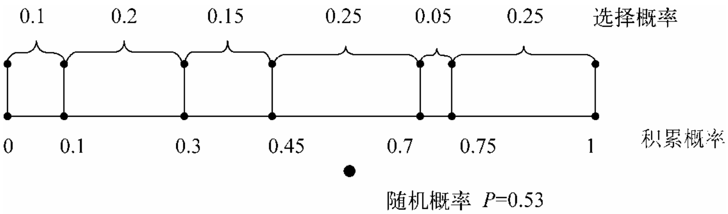


图 16-2 积累概率与选择概率的关系

假如选择算法随机产生概率 P=0.53，根据积累概率关系：$0.45 < P < 0.7$，于是第四个个体被选中。

在开始随机选择之前，种群中个体的选择概率和积累概率需要事先计算出来，计算的依据就是适应度函数给出的适应度评估值和种群的适应度总和：

```
double lastCf = 0.0;
//计算个体的选择概率和累积概率
for(i = 0; i < POPULATION_SIZE; i++)
{
    pop[i].rf = (double)pop[i].fitness / totalFitness;
    pop[i].cf = lastCf + pop[i].rf;
    lastCf = pop[i].cf;
}
```

轮盘赌模拟算法每次生成一个在 0 和 1 之间的随机数，然后与个体的积累概率比较，确定随机数位于哪个个体的积累概率区间就选择哪个个体，如果随机数小于第一个个体的积累概率，则选择第一个个体。具体的选择算法如下：

```
for(i = 0; i < POPULATION_SIZE; i++)
{
    double p = (double)rand() / (RAND_MAX + 1);
```

```
        if(p < pop[0].cf)
        {
            newPop[i] = pop[0];
        }
        else
        {
            for(int j = 0; j < POPULATION_SIZE; j++)
            {
                if((p >= pop[j].cf) && (p < pop[j + 1].cf))
                {
                    newPop[i] = pop[j + 1];
                }
            }
        }
    }
```

pop 是当前种群，newPop 是经过选择的下一代种群，至此，选择算子的算法实现就介绍完了，接下来介绍交叉算子的算法实现。

16.2.4　交叉算子设计

交叉算子采用的是多点交叉的策略，对两个随机选中的个体的基因进行交换，基因交换的位置和个数都是随机选择，使得新个体的基因更具随机性。交叉选择受交叉概率的控制，对种群中的每个个体生成一个 0 ~ 1 的随机数，判断这个随机数是否小于交叉概率，当小于交叉概率时，则选择这个个体参与基因交叉运算。个体选择的算法如下：

```
void Crossover(GATYPE *pop)
{
    int first = -1;//第一个个体已经选择的标识

    for(int i = 0; i < POPULATION_SIZE; i++)
    {
        double p = (double)rand() / (RAND_MAX + 1);
        if(p < P_XOVER)
        {
            if(first < 0)
            {
                first = i; //选择第一个个体
            }
            else
            {
                ExchangeOver(pop, first, i);
                first = -1;//清除第一个个体的选择标识
            }
        }
    }
}
```

first 变量是一个标识，用于判断是否已经选择过一个个体，P_XOVER 是交叉概率。交叉算法每选择两个个体后，就调用 ExchangeOver()函数进行基因交换。ExchangeOver()函数首先选择一个 1 ~ 7 的随机数作为基因交换的位数，然后对基因的每一位进行平均概率交换。正如你看到的那

样，交叉算法对基因交换后的合法性没有判断，这个工作已经由适应度评估函数和选择算子代劳。

```
void ExchangeOver(GATYPE *pop, int first, int second)
{
    /*对随机个数的基因位进行交换*/
    int ecc = rand() % OBJ_COUNT + 1;
    for(int i = 0; i < ecc; i++)
    {
        /*每个位置被交换的概率是相等的*/
        int idx = rand() % OBJ_COUNT;
        int tg = pop[first].gene[idx];
        pop[first].gene[idx] = pop[second].gene[idx];
        pop[second].gene[idx] = tg;
    }
}
```


16.2.5 变异算子设计

变异算子采用的是均匀变异的策略，对基因编码的每一位以平均分布的概率进行选择。当然，变异算子受变异概率的控制，以较低的概率选择进行变异的个体：

```
void Mutation(GATYPE *pop)
{
    for(int i = 0; i < POPULATION_SIZE; i++)
    {
        double p = (double)rand() / (RAND_MAX + 1);
        if(p < P_MUTATION)
        {
            ReverseGene(pop, i);
        }
    }
}
```

P_MUTATION 是变异概率，只有随机数小于变异概率时，才调用 ReverseGene()函数进行基因的变异处理。ReverseGene()函数首先选择一个 1 ~ 7 的随机数作为基因变异的变异点个数，然后使用一个均匀分布的随机数决定对基因中的哪些位进行变异。对于本例的基因编码，变异的方法就是 1 变成 0，0 变成 1。以下就是 ReverseGene()函数的实现代码：

```
void ReverseGene(GATYPE *pop, int index)
{
    /*对随机个数的基因位进行变异*/
    int mcc = rand() % OBJ_COUNT + 1;
    for(int i = 0; i < mcc; i++)
    {
        /*每个位置被交换的概率是相等的*/
        int gi = rand() % OBJ_COUNT;
        pop[index].gene[gi] = 1 - pop[index].gene[gi];
    }
}
```

16.2.6 这就是遗传算法

现在回顾一下图 16-1，流程图中所有的遗传算法关键元素都已经具备了，现在是展示遗传算法的真面目的时候了，下面就是遗传算法的主体代码：

```
GATYPE population[POPULATION_SIZE + 1] = { 0 };
Initialize(population);

    int totalFitness = EnvaluateFitness(population);
    for(int i = 0; i < MAX_GENERATIONS; i++)
    {
          Select(totalFitness, population);
          Crossover(population);
          Mutation(population);
          totalFitness = EnvaluateFitness(population);
    }
```

Initialize()函数负责种群基因初始化，很简单，就是生成一些随机数。将适应度评估函数和三个遗传算子的实现按照图 16-1 的流程图组织起来，就是遗传算法。当主体代码中的 for 循环结束以后，种群 population 中适应度最高的那个个体就是遗传算法的解。根据基因编码的规则，解码出背包问题的物品选择状态即可得到问题空间的解。

将第 3 章介绍的背包问题的数据代入遗传算法中，种群大小取值为 32，进化代数 T 取值为 500，交叉概率 P_c 取值为 0.8，变异概率 P_m 取值为 0.15。运行算法得到最终种群中最好的结果是适应度为 170 的个体（不止一个），其基因编码是[1,1,0,1,0,1,1]，转换成问题空间的解就是背包中选择编号为 1、2、4、6、7 的物品，能获得最大价值是 170，这和我们用其他算法得到的最优结果完全一致。

16.3 总结

遗传算法是一种带启发性的随机搜索算法，它不像传统的搜索算法那样，从单个值开始迭代搜索，按照特定的搜索顺序对整个解空间进行遍历。遗传算法的优点就是一开始就从一大群解中开始搜索，覆盖区域大，有利于找到全局最优解。

遗传算法通过基因的变化隐含地对解空间的一部分重点区域进行搜索，从问题的多个解开始并行搜索，对重点区域的选择是通过适应度函数和选择算子的运算来实现的，这也是启发性的体现。所以，不要对遗传算法过分崇拜，这只是一种搜索算法而已，只是比漫无目的的穷举搜索算法“聪明”那么一点而已，通过较小的计算量获得较大的收益。也不要以为遗传算法是高效算法，只要能用解析的方法直接得到最优解的问题，都不要试图用遗传算法，因为它比穷举搜索高明不了多少。

轮盘赌算法是各种随机化算法中常用的随机选择算法，原理简单，实现也简单，但是也存在致命的问题。假如一些选择概率非常小的个体连续出现，就会导致它们集中在一起在“赌轮”上占据一块很大的扇区，那么这块扇区就比较容易被选中，但实际上选择的都是选择概率非常小的

个体，这也是轮盘赌算法选择误差比较大的原因，在设计算法的时候需要注意这一点。

本章给出的算法是遗传算法的一种极其简单的实现，但是麻雀虽小，五脏俱全，可以作为今后更复杂的遗传算法设计的基础。我对这个算法做了一些评估，每批进行 500 次遗传算法模拟，连续进行多个批次，发现当进化代数 T 取 100 时，每批次平均有 450 ~ 460 次模拟算法能得到最优解；当 T 取 200 时，每批次平均有 480 ~ 490 次模拟算法能得到最优解；当 T 取 500 时，每批次能得到最优解的次数平均超过 495 次。对于这几十行代码来说，结果还不错，当然，它还有进一步优化的余地，有兴趣的读者自己动手吧。

16.4 参考资料

[1] Goldberg D E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989

[2] 维基百科：http://zh.wikipedia.org/wiki/遗传算法

[3] 徐宗本，张讲社，郑亚林. 计算智能中的仿生学：理论与算法. 北京：科学出版社，2003

[4] 陈国良，王煦法，庄镇泉. 遗传算法及其应用. 北京：人民邮电出版社，2001

[5] Koza J R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992

[6] Poli R, Langdon W, McPhee N. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008

第 17 章 计算器程序与大整数计算

几乎每个学习编程的同学都写过计算器程序，有可能是老师布置的作业，也有可能是自己的兴趣，总之，都会有一个自己版本的计算器程序。不管是只支持加减乘除四则运算的简单程序，还是支持括号、对数和乘方计算的复杂程序，都会面临一个问题，就是字长问题。简单地说，32 位的整数计算最大结果只能表示到 4294967295，两个超过 65535 的数字相乘就会溢出。但是 Windows 的计算器程序却能超过这个限制，这很令人沮丧，但是这背后的秘密其实就是大整数计算。

17.1 哦，溢出了，出洋相的计算器程序

我记得我写的第一个计算器程序。我花了一晚上的时间，还引入了逆波兰表达式，支持带括号的四则运算。我向同学们炫耀这个成果，其中一个同学输入了两个数相乘，结果我的程序可耻地打印出了一个不着边际的负数。我很快找到了问题的原因，用 C 语言的 `int` 类型能表示的最大整数是 2147483647，这个结果显然是溢出了。于是我很快修改了程序，用 `double` 代替 `int`，暂时解决了整数计算溢出问题。但是很快，我的同学就发现，我的计算器计算的结果和科学计算器计算的结果有偏差。无论我怎么调整代码，结果总是不准确，最后我只好放弃了。

后来我研究了一下 `double` 类型浮点数的定义，才明白是浮点数有效数字不足造成的结果不准确。IEEE 定义 `double` 类型的浮点数，是 1 个符号标志位，11 个指数位（包括一个指数符号位），另外 52 位是有效数字。52 位二进制有效数字，从十进制表示也就是 14 ~ 15 位数字，再长的数字就不能表示了。也就是说从数据输入时就被截断了，结果自然就不准确了。

要提供更大范围的整数计算，或提供更高的计算精度，原生的数据类型肯定不能满足要求了，必须使用大整数计算。比如圆周率 π，平常数学计算可能小数点后精确到六七位就足够了，但是对于天文计算，必须精确到上万位，或者更高的精度，否则计算几十亿光年外的星系的运行轨迹就会产生很大的误差。本章就来简单介绍一下大数计算的原理，并给出大数的加减乘除四则运算、乘方和求余的算法实现。这些算法在第 18 章介绍 RSA 算法时也会用到。

17.2 大整数计算的原理

任何一本关于计算机组成原理的书都会介绍计算机处理加减乘除运算的原理，以及运算器的实现方法。大数计算的方法也是采用这些原理，只不过不是使用逻辑电路实现各种运算器。同时，大数计算也会根据自身的特点，利用竖式手工计算的一些方法，用程序的方式在计算机中模拟这些方法，实现大数计算。

在开始介绍算法原理之前，先要确定大数的存储方式，也就是大数在计算机内的表示方式。一般来说，常见的大数存储方式有字符串和大数数组两种方式。字符串存储方式采用由数字组成的字符串存储和表示大数，比如“1230948437372666438483287276”。字符串存储方式的优点是比较直观，处理用户输入和输出都很简单，不需要额外的转换操作。但是字符串存储方式的缺点也很明显，计算时需要逐位将数字字符转换成数字进行计算，然后再将结果转换成该位对应的数字字符，这使得计算效率很低。采用大数数组的优点就是计算效率高，而且可以采用任意进制的数字存储，比如 2^8 进制，2^{16} 进制等。此外，存储效率也比较高，占用空间小。当然，采用大数数组存储方式的缺点就是不直观，处理用户输入和输出时需要进行字符串转换操作。

现在比较流行的几个大数运算库，基本都是使用大数数组方式存储和表示大数，本章介绍的大数算法也采用了这种方式。采用数组方式存储大数，一般采用每个数组元素表示“一位”的方式，数组元素从低到高分别表示大数每“一位”数字。每个数组元素对应大数的“一位”数字，便于进位和借位，计算过程中数字按位对齐也很简单。现在来介绍一下大数的“位”的概念，大数的每一位数字其实和十进制数字的每一位数字是一样的，区别仅仅是这一位数字能表示的计数单位个数。十进制数字每个数位可以用 0 ~ 9 表示 10 个计数单位，超过 10 就需要进位。同样，十六进制数字每个数位可以用 0 ~ 9，A ~ F 表示 16 个计数单位，超过 16 同样需要进位。现在，我们让每一位表示更多的计数单位，比如 256(2^8)进制，为了能表示 0 ~ 255 共 256 个计数，这个数位至少需要 8 个比特（刚好可以用 `unsigned char` 类型的数组表示）。256 进制的数字 10 表示 256 个计数单位，相当于十进制数字的 256。

对于 32 位体系的计算机系统来说，CPU 处理 32 位数据的效率比处理 8 位（单字节）数据高，因此，如果进一步扩展，采用 2^{32} 进制（0 × 100000000 进制），刚好可以用 `unsigned int` 类型表示大数的每一位，使得计算和存储都很高效。2^{32} 进制原理也很简单，就是用每一个 32 位整数表示大数的一个数位，比如 2^{32} 进制的数字 10 是“两位数”，用十进制表示就是 4294967296，如下所示：

2^{32} 进制的数字 10 表示为：00000001 00000000 = 4294967296（十进制）

高位　　低位

前面举例用的大数字 1230948437372666438483287276，用 2^{32} 进制的数字表示也是两位数：

8037F94B 71B59CEC = 1230948437372666438483287276（十进制）

高位　　低位

本章介绍的大数算法就采用了 2^{32} 进制表示大数，在开始介绍算法实现之前，我们先给出大

数的数据定义：

```
class CBigInt
{
    …
    //符号位，0 表示正数，1 表示负数
    unsigned int m_Sign;
    //大数在 x100000000 进制下的数字位数
    unsigned m_nLength;
    //用数组记录大数在 x100000000 进制下每一位的值
    unsigned long m_ulValue[MAX_BI_LEN];
    …
};
```

CBigInt 类没有使用的 m_ulValue 数组的最高位作为符号标志（有一些大数库的实现确实是这么做的），而是使用一个独立的属性 m_Sign 作为符号标志。这样做的好处是，在不考虑符号位的情况下，可以将 CBigInt 对象当作无符号数计算。另外，符号位单独控制，也带来了很多灵活性。比如，一个正数加上一个负数的情况，不必像计算机那样转换成补码进行计算，只需要将其转化成无符号数减法，然后再设置一下符号标志即可。CBigInt 类的内部结构就是先实现无符号数的加减乘除算法，然后再加上符号位的处理，支持带符号数的运算。

17.2.1　大整数加法

大整数运算的原理，就是用基本数据类型模拟大整数的加减乘除运算，包括进位和借位。在大数四则运算中，加法是最基本的运算，也最容易实现。大数加法的算法就是按位相加，只要处理好进位就行了。处理进位的关键是如何判断“溢出”是否发生，两个正数相加，如果大于 2^{32} 所能表示的最大正数，就发生溢出，溢出意味着要进位。如果没有 64 位整数类型，要判断两个 32 位整数相加是否发生溢出还真不容易，需要判断 CPU 的进位标志。但是有 64 位整数类型就简单了，直接将两数之和赋值给一个 64 位整数，然后判断是否大于 0×FFFFFFFF，如果大于 0×FFFFFFFF，就说明需要进位。

下面就以竖式加法演示一下大数加法的过程，如图 17-1 所示，这个过程和十进制竖式加法一样：

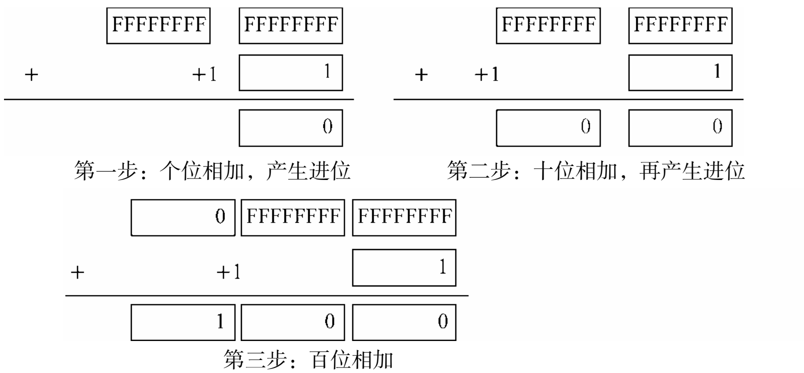


图 17-1　大数加法计算过程

和十进制的 99+1=100 一样，这两个大数的和也是 100，不过是 2^{32} 进制的 100，相当于十进制的 18446744073709551616。在不考虑符号位的情况下，我们先实现无符号数的大数加法：

```
void CBigInt::Add(const CBigInt& value1, const CBigInt& value2, CBigInt& result)
{
    result = value1;

    unsigned carry = 0;
    /*先调整位数对齐*/
    if(result.m_nLength < value2.m_nLength)
        result.m_nLength = value2.m_nLength;
    for(unsigned int i = 0; i < result.m_nLength; i++)
    {
        unsigned __int64 sum = value2.m_ulValue[i];
        sum = sum + result.m_ulValue[i] + carry;
        result.m_ulValue[i] = (unsigned long)sum;
        carry = (unsigned)(sum >> 32);
    }
    //处理最高位，如果当前最高位进位 carry!=0，则需要增加大数的位数
    result.m_ulValue[result.m_nLength] = carry;
    result.m_nLength += carry;
}
```


CBigInt::Add()函数将大数 value1 与 value2 的和存入 result，CBigInt::Add()函数不改变 value1 与 value2 的值，也不考虑符号位。要支持符号位其实很简单，如果参与计算的两个数符号相同，则直接调用 CBigInt::Add()函数计算两数之和，然后将符号标志设置为和两个数一样的符号即可。如果两个数的符号位不相同，则调用下一节将介绍的无符号减法函数，用两数中较大的数减较小的数，然后将结果的符号标志设置成与较大的数的符号标志一样。CBigInt 类重载了+运算符，用于支持带符号数的加法，根据上述描述，算法实现如下：

```
CBigInt CBigInt::operator+(const CBigInt& value) const
{
    CBigInt r;

    if(m_Sign == value.m_Sign)
    {
        CBigInt::Add(*this, value, r);
        r.m_Sign = m_Sign;
    }
    else
    {
        if(CompareNoSign(value) >= 0)
        {
            CBigInt::Sub(*this, value, r);
            r.m_Sign = m_Sign;
        }
        else
        {
            CBigInt::Sub(value, *this, r);
            r.m_Sign = value.m_Sign;
        }
```

```
    }

    return r;
}
```

17.2.2 大整数减法

和大整数加法一样，大整数减法的设计也从无符号数的减法开始。大数的减法是相对比较简单的算法，和加法一样，也是按位相减，只要处理好借位就可以了。当被减数当前位比减数小的时候，就要向前一位（高位）借位。“借”，十进制数字每一位数字有 10 个计数单位，因此，向前一位“借 1”相当于借了 10 个计数单位。2^{32} 进制每一位数字则有 0×100000000（2^{32}）个计数单位，因此，向前一位“借 1”相当于借了 0×100000000 个计数单位。

大数减法的算法流程和大数加法一样，从低到高按位计算。如果被减数的对应位上的数字大于或等于减数，则直接对这一位做减法计算，计算得到的值就是最终结果中这一位的值。如果被减数对应位上的数字小于减数，则设置借位标志，并从前一位“借 1”。当前一位的数字进行计算时，被减数除了减去减数，还要根据借位标志判断是否需要再减 1。现在我们仍然先给出无符号数减法的实现代码：

```
void CBigInt::Sub(const CBigInt& value1, const CBigInt& value2, CBigInt& result)
{
    CBigInt r = value1;

    unsigned int borrow = 0;
    for(unsigned int i = 0; i < r.m_nLength; i++)
    {
        if((r.m_ulValue[i] > value2.m_ulValue[i])||((r.m_ulValue[i] == value2.m_ulValue[i])&&(borrow
            == 0)))
        {
            r.m_ulValue[i] = r.m_ulValue[i] - borrow - value2.m_ulValue[i];
            borrow = 0;
        }
        else
        {
            unsigned __int64 num = 0x100000000 + r.m_ulValue[i];
            r.m_ulValue[i] = (unsigned long)(num - borrow - value2.m_ulValue[i]);
            borrow = 1;
        }
    }
    while((r.m_ulValue[r.m_nLength - 1] == 0) && (r.m_nLength > 1))
        r.m_nLength--;

    result = r;
}
```

CBigInt::Sub()函数的计算过程不考虑符号位，且假设被减数（value1）总是大于或等于减数（value2）。CBigInt::Sub()函数的实现做这个限制是有目的的，首先就是这样做简化了算法实现，是代码专注于按位减和借位的处理逻辑，避免一些不必要的判断和处理逻辑。其次，这个假设使

得大数的符号标识独立，可以用加法来模拟符号相异（两个数一正一负）的两个数的减法，避免了像计算机那样复杂的转码处理。

带符号的大数减法也可以由无符号大数的加法和减法模拟实现。其原理也很简单，如果被减数和减数符号相异，则调用 CBigInt::Add()函数在忽略符号标识的情况下计算二者之和，然后将符号位设置成和被减数符号位一样即可。如果被减数和减数符号相同，则比较二者的绝对值，用绝对值大的数减绝对值小的数，并将减过的符号标识设置成和绝对值大的那个数一致。CBigInt 类重载了“-”运算符，用于支持带符号数的减法，根据上述描述，算法实现如下：

```
CBigInt CBigInt::operator-(const CBigInt& value) const
{
    CBigInt r;

    if(m_Sign != value.m_Sign)
    {
        CBigInt::Add(*this, value, r);
        r.m_Sign = m_Sign;
    }
    else
    {
        if(CompareNoSign(value) >= 0)
        {
            CBigInt::Sub(*this, value, r);
            r.m_Sign = m_Sign;
        }
        else
        {
            CBigInt::Sub(value, *this, r);
            r.m_Sign = (m_Sign == 0) ? 1 : 0; //需要变号
        }
    }
```

17

CompareNoSign()函数是比较两个大数的大小，并忽略符号标识，相当于比较两个大数的绝对值。CompareNoSign()函数的实现很简单，如果两个数的位数不一样，则位数多的数比较大。如果两个数的位数相同，则从高位开始逐位比较。CompareNoSign()函数实现简单，此处就不再列出代码。

17.2.3 大整数乘法

大数乘法比大数加法和大数减法复杂一点，但是计算过程依然是按位相乘，并处理进位。乘法计算的进位处理和加法不太一样，加法的进位一般是“进 1”，但是乘法的进位可不一定是 1，但是也不会超过 2^{32}。观察一下十进制乘法的竖式计算过程，可以发现其主要计算过程就是乘数与被乘数按位相乘，处理进位，然后乘数和被乘数移位，重复上述过程，直到结束。大数的乘法计算可以仿照十进制乘法的计算过程实现，以三位的大数“1 5F 7FFFFFFF”乘以“F”为例，其竖式乘法计算过程如图 17-2 所示。

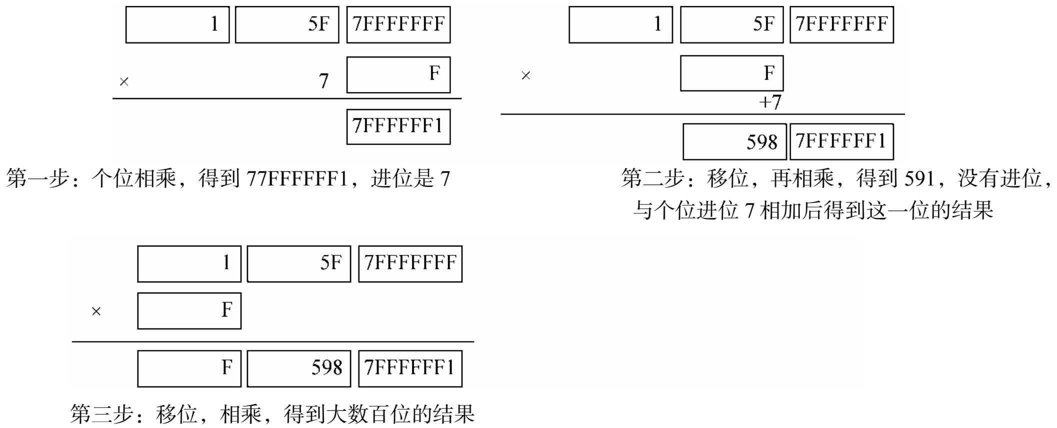


图 17-2 大数乘法计算过程

CBigInt::Mul()函数计算两个大数的乘积，这个函数不考虑符号位，只计算无符号大数的乘法。如果考虑带符号大数的乘法，逻辑上比带符号的加减法还简单，因为符号标识的确定非常简单：如果乘数和被乘数同号则结果为正数，否则结果为负数。CBigInt 类重载了*运算符计算带符号数的乘法，代码非常简单，此处不再列出代码。

```
void CBigInt::Mul(const CBigInt& value1, const CBigInt& value2, CBigInt& result)
{
    unsigned __int64 carry = 0;
    result.m_nLength = value1.m_nLength + value2.m_nLength - 1; //初步估算结果的位数
    for(unsigned int i = 0; i < result.m_nLength; i++)
    {
        unsigned __int64 sum = carry;
        carry = 0;
        for(unsigned int j = 0; j < value2.m_nLength; j++)
        {
            if(((i - j) >= 0)&&( (i - j) < value1.m_nLength))
            {
               unsigned __int64 mul = value1.m_ulValue[i - j];
               mul *= value2.m_ulValue[j];
               carry += mul >> 32;
               mul = mul & 0xffffffff;
               sum += mul;
            }
        }
        carry += sum >> 32;
        result.m_ulValue[i] = (unsigned long)sum;
    }
    if(carry != 0) //最后仍有进位，则大数位数需要扩大
    {
        result.m_nLength++;
        result.m_ulValue[result.m_nLength - 1] = (unsigned long)carry;
    }
}
```

17.2.4　大整数除法与模

除法表达的是两个数之间的倍数关系，这种倍数关系可以用连续的减法进行测试。大整数除法最简单的实现方法就是用被除数去减除数，重复这个减法过程，直到被除数小于除数为止，这个过程中进行减法的次数就是最终的结果。但是，这种方法也是效率最低的方法，如果除数非常小，而被除数又非常大，那么这个减的过程将非常耗时。必须对其进行优化，否则，任何有进取心的大整数计算都不会采用这种方法实现除法运算。

优化的方向就是“试商”。这和我们做竖式除法的原理一样，我们先用 135 ÷ 6 演示一下十进制竖式除法的过程，如图 17-3 所示。

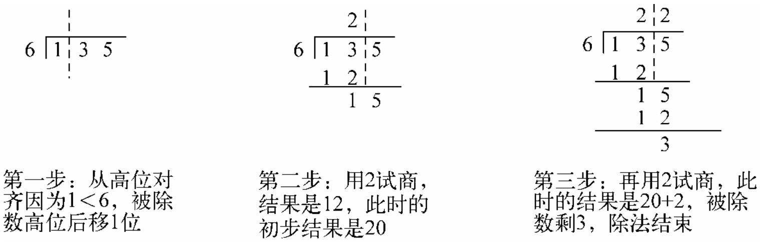


图 17-3　十进制除法计算过程

除法的整个过程仍然是多次减法重复的过程，但是试商的结果是每次可以减去除数的若干倍，能极大地加快这个减的过程，提高计算效率。

大整数的除法，依然采用这个原理，从被除数和除数的高位开始，如果被除数的高位小于除数，则用被除数的高位和次高位组成两位大数与除数的高位做除法（这要得益于系统提供的 64 位整数原生运算支持，大大简化了算法的复杂度）。这个除法的结果就是试商的依据，同时也是结果的一部分，要累加到最终的结果中。但是，到底累加多少呢？假如这个除法的结果是 5，那么，结果是加 5 还是加 50，或者是 500、5000 呢？那就要看被除数拿掉最高位和次高位后还剩多少位，对于大整数来说，如果是剩 0 位，则结果就累加 5，如果剩 1 位，则结果就累加 5×2^{32}，如果剩 2 位，则结果就累加 5×2^{64}，以此类推。

整数的除法运算通常都不会刚好除尽，其结果总是分成两部分：商和余数，大整数也不例外。被除数在整个除法过程中逐步减少，最终当被除数小于除数的时候，此时的被除数的值就是余数，因此，除法和取模是同一个过程。

`CBigInt::Div()`函数计算两个大数的除法，得到商和余数。这个函数同样不考虑大整数的符号，`CBigInt` 类重载了/和%运算符，用于提供带符号大整数的除法和取模。两个带符号位的大整数相除，其结果的符号位判别方法和乘法一样，如果两数同号，则商结果是正数，如果两数异号，则商结果是负数。对于取模运算，也就是余数的符号则更简单，它总是和被除数的符号一致。

```
void CBigInt::Div(const CBigInt& value1, const CBigInt& value2, CBigInt& quotient, CBigInt& remainder)
{
```

```
        CBigInt r = 0;
        CBigInt a = value1;
        while(a.CompareNoSign(value2) >= 0)
        {
            unsigned __int64 div = a.m_ulValue[a.m_nLength - 1];
            unsigned __int64 num = value2.m_ulValue[value2.m_nLength - 1];
            unsigned int len = a.m_nLength - value2.m_nLength;
            if((div == num) && (len == 0))
            {
                CBigInt::Add(r, CBigInt(1), r);
                CBigInt::Sub(a, value2, a);
                break;
            }
            if((div <= num) && (len > 0))
            {
                len--;
                div = (div << 32) + a.m_ulValue[a.m_nLength - 2];
            }
            div = div / (num + 1);
            CBigInt multi = div; //试商的结果
            if(len > 0)
            {
                multi.m_nLength += len;
                unsigned int i;
                for(i = multi.m_nLength - 1; i >= len; i--)
                    multi.m_ulValue[i] = multi.m_ulValue[i - len];
                for(i = 0; i < len; i++)
                    multi.m_ulValue[i] = 0;
            }
            CBigInt tmp;
            CBigInt::Add(r, multi, r);
            CBigInt::Mul(value2, multi, tmp);
            CBigInt::Sub(a, tmp, a);
        }
        quotient = r;
        remainder = a;
    }
```

17.2.5 大整数乘方运算

整数的乘方运算可以分解为整数的连乘，因此乘方的最简单实现方法就是用连续乘法计算代替。采用这种实现方案，计算一个数的 n 次方需要做 n 次乘法计算。思考一下，有没有方法可以减少一些乘法计算次数？来看一个例子，计算 a^9，根据乘方的意义：$a^9 = a^8 \times a$，只要能计算出 a^8，则计算 a^9 只需要做一次乘法。a^8 又可以分解为 $a^4 \times a^4$，因此，只需要计算出 a^4，a^8 也只需要一次乘法就可以得到。继续这个过程，a^4 又可以分解为 $a^2 \times a^2$，只要计算出 a^2，则只需要一次乘法计算就可以得到 a^4。最后计算 a^2 也只需要一次乘法，看到了吧，整个过程变成了 4 次乘法计算，这就是神奇的平方–乘降幂法，也称为二进制平方和乘法。利用乘方计算的数学性质将其逐步分解，可以有效地减少乘法计算量，提高计算效率。

计算乘方的算法实现刚好是上述分析过程的逆过程，仍以计算 a^9 为例，设临时变量 t 初始化为 a，结果 r 初始化为 1。乘方 9 的二进制表示是 1001，从最低位开始处理。最低位是 1，计算 r=r*t，同时计算 $t=t^2$，此时 r=a，$t=a^2$。倒数第二位是 0，计算 $t=t^2$，此时 $t=a^4$。倒数第三位仍然是 0，继续计算 $t=t^2$，此时 $t=a^8$。最高位是 1，计算 r=r*t，此时 $r=a*a^8=a^9$，完成计算。CBigInt::Power()函数计算 value 的 *n* 次方，结果存放在 result 中，这个函数也不考虑符号位。CBigInt 类重载了^运算符，这个重载^运算符的版本支持带符号数的乘方计算，处理符号标识的方法也很简单，如果 *n* 是奇数，结果的符号标识与 value 一样，如果 *n* 是偶数，结果的符号标识总是+号。

```
void CBigInt::Power(const CBigInt& value, const CBigInt& n, CBigInt& result)
{
    result = 1;
    CBigInt t = value;

    for(__int64 i = 0; i < n.GetTotalBits(); i++)
    {
        if(n.TestBit(i))
        {
            CBigInt::Mul(result, t, result);
        }
        CBigInt::Mul(t, t, t);
    }
}
```

17.3 大整数类的使用

至此，我们已经实现了大整数的四则运算和乘方计算（包括取模计算），这也是整数计算中常用的几种方法。用这些方法已经可以支撑我们完成比较复杂的大数计算，包括 RSA 加密算法需要的模幂和模乘运算。除此之外，为了实现类似计算器的功能，还必须有和用户交互的接口。大整数在计算机内部是以数组的形式存在，占用空间小，并且高效，但是如果反映在用户界面上，却不符合人类的使用习惯。通常人们还是习惯使用数字组成的字符串输入数字，也只看得懂字符串形式的数字（尽管当数字位数非常多的时候，人们已经不知道其实际意义了）。

大整数的输入，是将数字组成的字符串转换成内部的数组形式的大数，通常这些字符串都是十进制的数字，但是也可以是十六进制的数字。把一个数字字符串转换成一个大数与转换成普通的整数本质上是一样的，都可以参照 C 语言的库函数 atoi()来实现。同样，大整数的输出也可以参照 itoa()函数实现。

17.3.1 与 Windows 的计算器程序一决高下

如果早点有了 CBigInt 类，就不会在同学面前出丑了，现在回过头再看看 Windows 自带的计算器程序，发现其整数计算最大也只能支持到 18446744073709551615，真是太弱了。现在向小伙伴们炫耀一下吧，谁能计算出：

1445628445840946744607609235783709524795667819061352

乘以

18446445674407667997096551006152135

的结果是多少？当然是：

26666706591583411822193194835830799779242590142029250059182106110878932434973010786520

是否正确？现在只有你知道。

17.3.2 最大公约数和最小公倍数

求两个数或多个数的最大公约数（greatest common divisor）和最小公倍数（least common multiple）是整数计算中最常见的算法问题，这里我们来讨论几种计算最大公约数和最小公倍数的方法。首先看看最大公约数，求解最大公约数有很多算法，比如辗转相除法、辗转相减法以及小学生都会的短除法等。辗转相除法在汉代的《九章算术》一书中就有记载，在西方又被称为欧几里得算法，是求最大公约数的传统算法，这种方法进行辗转相除的理论依据是下面的定理：

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(b, a \bmod b) \qquad （a \bmod b \text{ 表示 } a \text{ 除以 } b \text{ 的余数}）$$

这就是朴素欧几里得定理，利用这个定理实现最大公约数的递归算法非常简单。在一些不适合使用递归算法的场合（比如某些单片系统），也可以使用非递归的算法，求最大公约数算法的大数计算版本一般采用非递归的算法实现：

```
CBigInt EuclidGcd(const CBigInt& a, const CBigInt& b)
{
    CBigInt c = (a > b) ? a : b;
    CBigInt result = (a > b) ? b : a;

    c = c % result;
    while(c != 0)
    {
        CBigInt tmp = c;
        c = result;
        result = tmp;
        c = c % result;
    }

    return result;
}
```

辗转相除法实现简单，效率还可以，但是大数求余需要大数除法的支持，而大数除法一般效率不高，如果能够避免除法和取模，则可以极大地提高求最大公约数算法的效率。J.Stein 在 1961 年提出了一种改进的算法，只使用大数的加减法和移位（除 2 可用移位代替），这称为 Stein 算法。Stein 算法的理论依据是：

```
GCD(ka, kb) = k * GCD(a, b)
```

当 k 取特别的值 2 时，可以利用整数的移位操作递归地对原数据 a 和 b 进行规约。下面来看看 Stein 算法的概念实现：

```
CBigInt SteinGcd(const CBigInt& a, const CBigInt& b)
{
    CBigInt biger = (a > b) ? a : b;
    CBigInt smaller = (a > b) ? b : a;

    if(smaller == 0)
        return biger;
    if((biger % 2 == 0) && (smaller % 2 == 0))
        return SteinGcd(biger / 2, smaller / 2) * 2;
    if(biger % 2 == 0)
        return SteinGcd(biger / 2, smaller);
    if(smaller % 2 == 0)
        return SteinGcd(biger, smaller / 2);

    return SteinGcd((biger + smaller) / 2, (biger - smaller) / 2);
}
```


看起来好像比传统算法有更多的除法和乘法，但是取模可以用位测试代替，乘 2 和除 2 都可以用整数移位来代替，实际上是规避了效率比较低的大数除法，这个给出的 SteinGcd()函数只是一个概念实现，有兴趣的读者可以用移位对其进行优化。

除了辗转相除法和 Stein 算法，*辗转相减法*也是一种比较容易编程实现的算法。辗转相减法看起来只用了减法，避免了乘法和除法，但是实际效果并不理想，特别是在两个数相差很大的情况下，会导致循环很多次也无法收敛。这里只给出算法，读者可自行研究。

```
CBigInt SubstractGcd(const CBigInt& a, const CBigInt& b)
{
    CBigInt aa = a;
    CBigInt bb = b;

    while(aa != bb)
    {
        if(aa > bb)
        {
            aa = aa - bb;
        }
        else
        {
            bb = bb - aa;
        }
    }

    return aa;
}
```

几个数公有的倍数就是公倍数，其中最小的一个就是最小公倍数。求最小公倍数的方法也有很多，比如短除法、分解质因数法等，最简单的方法是利用最大公约数和最小公倍数的关系间接

地获得最小公倍数。以两个数为例，最小公倍数和最大公约数存在以下关系：

$$最大公因数 \times 最小公倍数 = 两数的乘积$$

根据这个关系得到简单的求最小公倍数的算法实现如下：

```
CBigInt GcdLcm(const CBigInt& a, const CBigInt& b)
{
    CBigInt r = (a * b) / EuclidGcd(a, b);

    return r;
}
```

这个算法存在的主要问题是大数的乘法，会导致需要超过一倍的存储空间存储大数的乘积，如果 CBigInt 类只能支持 2048 比特的大整数，则 GcdLcm()函数只能计算两个小于 1024 比特的大整数的最小公倍数。如果不使用最大公约数帮忙，还可以考虑用自加加整除测试的方法计算最小公倍数。假设要求整数 a 和 b 的最小公倍数，首先看 a 是否能被 b 整除，如果不能就继续测试 $2a$ 能否被 b 整除，继续这个过程，直到 na 的时候能被 b 整除，则 na 就是 a 和 b 的最小公倍数。用这种方法实现的算法如下：

```
CBigInt NormalLcm(const CBigInt& a, const CBigInt& b)
{
    CBigInt r = a;

    while(r % b != 0)
    {
        r += a;
    }

    return r;
}
```

很显然，这种方法也存在问题，比如 a 非常小而 b 非常大的时候，会导致 while 循环相当漫长。

17.3.3 用扩展欧几里得算法求模的逆元

对于任意整数 a、b 和 c，形如 $ax + by = c$ 的方程就被称为线性不定方程。根据贝祖定理[①]，如果 c 是 a 和 b 的最大公约数，则该不定方程存在整数解。当 c 是 a 和 b 的最大公约数的整数倍时，不定方程有多组解，每一组解之间存在 $c/\gcd(a,b)$的倍数关系。求解 $ax + by = \gcd(a,b)$通常可使用扩展欧几里得算法，其基本原理仍然是朴素欧几里得定理。

扩展欧几里得算法其实并不复杂，其推导过程和很简单。首先利用朴素欧几里得定理给出的最大公约数辗转关系：$\gcd(a,b) = \gcd(b,a\%b)$，将不定方程 $ax + by = \gcd(a,b)$转换成另一种形式：

$$ax + by = \gcd(a,b) = \gcd(b,a\%b) = bx' + (a\%b)y'$$

重复（递归）利用以上辗转关系，最终会有 $a\%b=0$，原方程最终可以转换成：$ax'+0*y'= \gcd(a,0)$，

① 贝祖定理：给两个整数 a 和 b，必然存在一对整数 x 和 y，使得 $ax+by=\gcd(a,b)$。

解这个方程很容易，只要令 $x'=1$，$y'=0$ 即可。很显然，经过这样的转换后得到的解，已经不是原方程的解，但是，根据上述转换的递推关系，可以反向递推出原方程的解。

$a\%b$ 可以理解为 $a-(a/b)*b$，将其代入到上述递推关系中，可以得到每一次辗转变换后(x,y)与(x',y')的递推关系：

$$ax + by = bx' + (a\%b)y' = bx' + (a - (a/b)*b)y' = ay' + b(x' - (a/b)y')$$

利用恒等关系，可以得到以下递推关系：

$$x = y'$$

$$y = x' - (a/b)*y'$$

利用这个关系逐级反推即可得到原方程的解。ExtEuclid()函数给出了求解形如 $ax + by = 1$ 的不定方程的算法实现（暗含 $\gcd(a,b)=1$ 的条件）：

```
CBigInt ExtEuclid(const CBigInt& a,const CBigInt& b,CBigInt& x,CBigInt& y)
{
    if(b == 0)
    {
        x = 1;
        y = 0;
        return a;
    }
    CBigInt xp,yp;
    CBigInt c = ExtEuclid(b, a%b, xp, yp);
    x = yp;
    y = xp - (a / b) * yp;

    return c;
}
```

17

除了求解线性不定方程，扩展欧几里得算法还被用来求解线性同余方程和模的逆元。我们定义以下形式的方程为同余方程：$ax \equiv b \pmod{n}$，当且仅当满足 $\gcd(a,n) \mid b$ 条件时，此方程有整数解，且有 $\gcd(a,n)$个整数解。如果引入一个整数 y（y 可为任意整数值），将同余方程的右边转换成 $ny + b$，则线性同余方程可以转换为线性不定方程 $ax - ny = b$，如此一来就可以利用扩展欧几里得算法求解 x（y 为指定整数，注意符号可能是反的）。对于同余方程 $ax \equiv b \pmod{n}$，若 $\gcd(a,n) = 1$，则方程有唯一的整数解，在这种情况下，如果 b 也等于 1，则这个唯一的整数解就被称为 a 对模 n 的乘法逆元，记为 $x = a^{-1}$。和求最大公约数一样，求大整数模的乘法逆元也是 RSA 非对称密钥加密体系中的一个基本操作，具有非常重要的意义。

当 $b = 1$ 的时候，同余方程 $ax \equiv 1 \pmod{n}$可以转化为线性不定方程：$ax - ny = 1$，这样就可以利用前面讨论的扩展欧几里得算法求解 x 和 y。需要注意的是，此时得到的 y 的符号是反的，但是同余方程的转换只是将 y 作为一个辅助整数引入，并不关心其值。前面已经分析过扩展欧几里得算法的求解步骤和解的递推计算方法，只需要用 n 代替 b，并忽略 y 的值，就可以得到同余方程的求解算法。

```
CBigInt CongruenceEquation(const CBigInt& a, const CBigInt& n)
{
    CBigInt x,y;

    CBigInt r = ExtEuclid(a, n, x, y);
    if(r > 0)
    {
        return x;
    }

    return 0;
}
```

17.4 总结

有人认为大数计算是科学家才会用到的东西，生活中不会用到这么大的数字，其实不然。大数计算可不仅仅是用来做计算器用的，在天文、物理等各个领域都离不开大数运算。著名的 RSA 算法的本质也是大整数的指数计算。在生物学领域，DNA 的分解和重组研究也离不开大数运算。

Miracl 和 Freelip 都是比较著名的大数计算库，本章的算法有些就是参考了这些库的设计思想。除此之外，很多专业的加密软件包都会包含大数计算的库，读者可自行研究。完整的 CBigInt 类的实现代码包含在本章附带的示例代码中，包括简单的测试用例。第 18 章介绍 RSA 算法时，还会用到这个类的实现。

17.5 参考资料

[1] 白中英. 计算机组成原理（第四版）. 北京：科学出版社，2008

第 18 章 RSA 算法——加密与签名

RSA 算法（Rivest-Shamir-Adleman）是非对称公钥加密体系的开山鼻祖，经过几十年的发展，RSA 算法在银行、军事、通信等领域得到了广泛的应用。RSA 算法不仅用于数据加密，还可用于数字签名和身份验证。虽然现在椭圆加密算法（Elliptic Curves Cryptography，ECC）的应用也是如日中天，但是 RSA 算法仍然在非对称公钥加密体系中占有一席之地。

RSA 算法是一种非常简洁的加密算法，远没有人们想象的那么复杂和神秘。RSA 算法背后的数学理论就是大素数分解难题，其算法实现的核心是大整数的模幂运算。有了第 17 章介绍的大整数运算基础，实现 RSA 算法就易如反掌。

18.1 RSA 算法的开胃菜

RSA 算法的核心是大整数的*模幂运算*（Modular Power），模幂运算又称为*模乘方运算*。用数学表达式表示模幂运算就是：

$$C = A \wedge B \pmod{n}$$

我们已经实现了大数的乘方运算和取模运算，只需要先计算 A 的 B 次方，然后再对这个中间结果用除法求余数就可以得到结果。但是这个方案存在一个很大的问题，就是乘方和除法取余数的计算量都非常大，效率不高。除此之外，乘方计算的中间结果将是一个非常大的数，因为操作数不确定，所以也无法估计这个结果会多大，大数必须支持非常多的位才能保存这个中间结果。

根据 RSA 算法的性质可以看出，模幂运算的性能决定了 RSA 算法的性能。为了解决模幂运算效率的问题，现代数学界提出了很多解决方案。这些解决方案的基本思想都是先将模幂运算转换成*模乘运算*（Modular Multiplication），然后再用高效的算法处理模乘运算。本节要介绍的快速模幂和模乘算法都是实现高效 RSA 算法必不可少的组件，可以称为 RSA 算法的开胃菜。

18.1.1 将模幂运算转化为模乘运算

前面介绍过，对于模幂运算，如果用先求幂再取模的方式直接计算结果，在很多情况下是不能接受的。即便不考虑乘方计算的低效率，中间结果的存储也是个棘手的问题。以 1024 比特的大整数的乘方计算为例，其二次方的结果最大可能需要 2048 比特的存储空间，其 1024 次方的结果最大可能需要 1 兆比特（约 128 千字节）的存储空间。对于大数计算来说，1024 作为指数简直就是个微不足道的值，考虑到指数也可能是 1024 比特的大整数，存储中间结果最终需要的内存将超出计算机的能力。

模幂计算的解决思路是将其转化为模乘计算，避免直接求幂带来的存储和效率问题。模乘的数学表达式是：

$$C = A \times B \ (\mathrm{mod}\ n)$$

那么，如何将模幂计算转化成模乘计算呢？在第 17 章介绍大整数计算时，提到过一种优化乘方运算的“平方–乘降幂法”，在计算大数乘方时可以有效地减少乘法计算的次数。在处理模幂运算时，同样可以利用这种思想将模幂运算转化成一些列模乘运算。将模幂运算转化成模乘运算，需要利用模运算的两个特性，即：

$$(a \times b)\%n = (a\%n \times b\%n)\%n$$

$$(a + b)\%n = (a\%n + b\%n)\%n$$

以计算 $a^9 \% n$ 为例，可以分解为$(a^8 \% n \times a \% n)\%n$，$a^8 \% n$ 又可以分解为$(a^4 \% n \times a^4 \% n)\%n$，$a^4 \% n$ 又可以继续分解为$(a^2 \% n \times a^2 \% n)\%n$，$a^2 \% n$ 最终分解为$(a \% n \times a \% n)\%n$。利用这种思想，$a^9 \% n$ 的模幂运算就转换成 5 次模乘运算。这种转换的算法实现类似于 CBigInt::Power()函数的实现，非常简单：

```
CBigInt ModularPower(const CBigInt& M, const CBigInt& E, const CBigInt& N)
{
    CBigInt k = 1;
    CBigInt n = M % N;

    for(__int64 i = 0; i < E.GetTotalBits(); i++)
    {
        if(E.TestBit(i))
        {
            k = (k * n) % N;
        }
        n = (n * n) % N;
    }

    return k;
}
```

CBigInt ModularPower()函数可以将 $M^E \% N$ 的计算转化成平均 3log(E)/2 次模乘运算。这只是对模幂运算优化的第一步，接下来还要利用蒙哥马利模乘再对模乘运算进行优化，化解不必要的除法计算，进一步提高模幂计算的速度。

18.1.2 模乘运算与蒙哥马利算法

影响模乘运算速度的关键在于费时的取模运算（除法计算），如果在模乘运算中不用除法或尽量少用除法，将大大提高模幂运算的速度。数学家们研究了很多快速计算模乘的算法，蒙哥马利算法（Montgomery Reduction）就是其中的一种。大家可能对二战时期著名的英国陆军元帅蒙哥马利比较熟悉，但是此蒙哥马利非彼蒙哥马利。蒙哥马利算法是由彼得·蒙哥马利（Peter L. Montgomery）在 1985 年提出的一种大数模乘快速计算方法，又称为蒙哥马利约分算法。

蒙哥马利约分的基本思想就是选择一个适当的 $R = 2^k$，k 满足条件：$n < 2^k$，将对 n 的取模运算转化为对 R 的完全剩余系计算，对 R 的除法计算可以转换为移位操作，从而避免了除法计算。因为 $R > n$，且 R 是 2 的整数幂，n 是素数，因此 R 和 n 互素，根据欧拉方程有解的条件可知，一定存在整数 $0 < R^{-1} < n$ 和 $0 < n' < R$，满足 $RR^{-1} - nn' = 1$。此时可以将 $A \times B \pmod{n}$的计算转化为计算 $A' \times B' \times R^{-1} \pmod{n}$，其中 A'和 B'分别是 A 和 B 对 R 的剩余系表达，即：

$$A' = A \times R \pmod{n}$$

$$B' = B \times R \pmod{n}$$

$A' \times B' \times R^{-1} \pmod{n}$称为蒙哥马利模乘，它可以利用蒙哥马利约分算法高效地计算出来，蒙哥马利约分算法的计算方法如下：

```
function REDC(A', B', n', R, N)
    S = A' × B'
    m = (S mod R) × n' mod R
    t = (S + mN) / R
    if(t >= N)
        then return t - N
        else return t
```

根据以上描述的方法可以很容易写出蒙哥马利约分算法的实现代码：

```
CBigInt MontgomeryReduction(const CBigInt& X, const CBigInt& Y, const CBigInt& Np, const CBigInt& N,
    const CBigInt& R)
{
    CBigInt S = X * Y;
    CBigInt m = (S * Np) % R;
    S = (S + m * N) / R;
    if(S >= N)
        return S - N;
    else
        return S;
}
```

因为 R 是 2 的整数幂，只需要将对 R 的取模和除法运算转化成移位运算，就可以得到真正高效的模乘算法。蒙哥马利约分算法需要为计算 R 的剩余系而付出一些额外的开销，因此对于单次模乘计算，蒙哥马利约分算法并没有优势，但是对于像模幂运算这样需要多次反复计算模乘的情况，使用蒙哥马利约分算法可以极大地提高模幂计算的速度。

18

18.1.3 模幂算法

现在可以使用蒙哥马利约分算法改造 18.1.1 节给出的模幂算法。首先要利用同余方程计算出 n'，然后再将模幂运算的底数 M 转换到 R 的剩余系，并用“平方–乘降幂法”逐次计算蒙哥马利模乘，最后将蒙哥马利模乘运算的结果转出 R 的剩余系，得到最终的结果。在选择 R 的时候，我们取 k 值为 32 的整数倍，这样在计算蒙哥马利约分算法的移位处理的时候，对于我们的大整数 CBigInt 来说，一次移动一个 unsigned int 大数位，速度更快。最后给出使用蒙哥马利算法优化后的模幂算法实现：

```
CBigInt ModularPower(const CBigInt& M, const CBigInt& E, const CBigInt& N)
{
    CBigInt R = 1;
    R <<= N.m_nLength * 32;

    CBigInt Np = CongruenceEquation(R - N, R);
    //转换到 R 的剩余系
    CBigInt Mp = (M * R) % N;
    CBigInt D = R % N;

    for(__int64 i = 0; i < E.GetTotalBits(); i++)
    {
        if(E.TestBit(i))
        {
            D = MontgomeryReduction(D, Mp, Np, N, R);
        }
        Mp = MontgomeryReduction(Mp, Mp, Np, N, R);
    }
    //转出 R 的剩余系
    D = MontgomeryReduction(D, 1, Np, N, R);

    return D;
}
```

18.1.4 素数检验与米勒–拉宾算法

素数在数论中是一个很大的分支，很多数学定理都和素数有关，有人甚至将其独立出来称为素论，可见素数对于数学的重要性。RSA 算法在生成密钥对时需要两个随机大素数，并将它们的乘积作为公共模数 n，这就需要有对应的素数生成算法。素数生成没有什么特殊方法，就是生成随机大数作为疑似素数，然后用素数检验方法检验是否是“真”的素数，如果是就返回结果，如果不是就继续上述过程。由此可见，要生成一个素数，必须要有一套判断素数的方法。1000 以内的小素数可以用素数的定义直接判断，大素数则要采用特定的算法进行素性测试。要进行素性测试，先来了解一下费马小定理，定义如下：

设 p 是素数，a 是任意整数，且 $a \,!\!\equiv 0 (\mathrm{mod}\, p)$，则 $a^{(p-1)} \equiv 1 (\mathrm{mod}\, p)$

一般来说，可以利用费马小定理直接进行素数测试，这就是费马测试（Fermat）。费马测试实际上是利用费马小定理的逆定理进行反向证明，不幸的是，费马小定理只是素数检验的必要条

件，其充分条件，也就是费马小定理的逆定理并不成立，因为存在卡米歇尔数[1]（Carmichael）。费马测试是个概率测试，并没有得到广泛的应用，目前判断大素数（特别是超过 512 位的大素数）普遍采用的方法是米勒–拉宾（Miller–Rabin）算法。

1975 年，卡内基梅隆大学计算机系的米勒（Gary Lee Miller）教授首先提出了基于广义黎曼猜想[2]的确定性算法，由于广义黎曼猜想并没有被证明，直接引用广义黎曼猜想就存在理论上的缺陷。所以以色列耶路撒冷希伯来大学的拉宾（Michael O. Rabin）教授对其进行了改进，提出了不依赖于该假设的随机化算法，这就是米勒–拉宾算法的由来。米勒–拉宾素性检验算法利用随机化算法判断一个数是合数还是可能是素数，请注意，这里用词是“可能”，因为米勒–拉宾算法也是一种判断素性的概率算法。虽然是概率算法，如果加上限制条件，米勒–拉宾算法也可以作为一种确定性算法。

用米勒–拉宾素性检验法检验算法判断 n 是否是素数，首先将 $n-1$ 分解为 $m*2^k$，然后在$[1, n-1]$区间上随机选一个整数 a，对于$[0, k-1]$区间中的每一个值 r，检测：

$$a^m (\bmod\ n) \neq 1 \text{ 和 } a^{m\times 2^r} (\bmod\ n) \neq -1$$

两个条件是否同时成立，如果两个条件同时成立，则 n 是一个合数，否则，n 有 75%的概率是一个素数。有此可知，做一次检验，即便 n 不满足两个成为合数的条件，仍然有 1/4 的可能性是合数。但是，如果用足够多的随机数 a 对其进行多次检验，则可以降低 n 是合数的可能性。假设检验次数是 t，则 n 是合数的可能性是 $P(c) = 1/4^t$。如果进行 5 次检验都符合上述情况，n 是合数的可能性就降到 0.098%，即 n 有 99.9%的可能是素数。如果进行 10 次检验，则 n 是素数的可能性就达到 99.9999%。用米勒–拉宾算法检验素数，一般至少需要检验 5 次，严格的场合可能需要检验更多的次数，比如 50 次。

`MillerRabinHelper()`函数是一次米勒–拉宾检验的算法实现，其中 m 和 k 两个参数需要实现计算出来，计算的方法将在 `MillerRabin()`函数中给出。根据检验规则的定义，需要计算 a 与 m 关于 n 的模幂，以及 a 与 $m\times 2^r$ 关于 n 的模幂，根据幂运算关系的特点，a 的 $m\times 2^r$ 次方与 a 的 m 次方存在平方的关系。如果令 $b = a^m$，则 a 的 $m\times 2$ 次方就是 b^2，则 a 的 $m\times 4$ 次方就是 b^4，以此类推。如果采用 b^2 的累积，可以减少很多计算量，因此，一般算法实现都会采用 b^2 的累积代替计算 a 的 $m\times 2^r$ 次方，`MillerRabinHelper()`函数也不例外。

---

① 卡米歇尔数：能满足费马小定理，但是又不是素数的数。

② 德国数学家黎曼在 1858 年写了一篇只有 8 页长的关于素数分布的论文，提出了著名的广义黎曼猜想（Riemanns Hypothesis）。这个猜想是指黎曼 ζ 函数（ζ 音：齐塔）：$\zeta(s) = \Sigma 1/n\^s$ (n 从 1 到无穷大)的非平凡零点都在 $\mathrm{Re}(s) = 1/2$ 的直线上（也就是说所有非平凡零点的实部都是 1/2）。看似简单的问题实际上并不容易，求多项式的零点，特别是求代数方程的复根都不是简单的问题。数学家把复平面上 $\mathrm{Re}(s) = 1/2$ 的直线称为临界线（Critical Line）。1914 年，英国数学家哈代（G.H. Hardy）首先证明这条临界线上有无穷个零点。三位荷兰数学家利用计算机对最初的两亿个 ζ 函数的零点进行检验，目前已经证明了 2/5 的复零点都在这条直线上，并且在这条直线之外至今还没有发现其他复零点。这初步证明黎曼的假设是对的，但是这个检验的过程还在继续，因此，广义黎曼猜想是对还是错还没有定论。

```
bool MillerRabinHelper(const CBigInt& a, const CBigInt& m, int k, const CBigInt& n)
{
    CBigInt b = ModularPower(a, m, n);

    if(b != 1)
    {
        for(int r = 0; r < k; r++)
        {
            if(b != (n - 1)) //b != 1 && b != n-1，满足合数条件
            {
                return false;
            }
            b = ModularPower(b, 2, n);
        }
    }

    return true;
}
```

MillerRabinHelper()函数返回 false 表示 n 不是素数，返回 true 表示 n 有 75%的可能是一个素数。当 MillerRabinHelper()函数返回 true 时，需要使用新的随机数 a 对 n 继续检验，直到满足检验次数条件。MillerRabin()函数首先根据 n 计算出 m 和 k，然后多次调用 MillerRabinHelper()函数进行检验。产生随机数 a 的时候，总是选一个 32 位以内的小随机数，目的是减少检验的计算量。a 的二进制位数总是比 n 少一位，且最大不超过 32 位，保证 a 总是小于或等于 $n-1$。

```
int MillerRabin(const CBigInt& n)
{
    CBigInt m = n - 1;
    int k = 0;

    //根据 n-1 = m*2^k，计算 m 和 k
    while(!m.TestBit(0))
    {
        m >>= 1; //m = m / 2;
        k++;
    }

    CBigInt a,b;
    for(int i = 0; i< M_R_TEST_COUNT; i++)
    {
        __int64 nbits = n.GetTotalBits();
        // 1 <= a <= n - 1
        a = CBigInt::GenRandomInteger((nbits > 32) ? 32 : nbits - 1) + 1;
        if(!MillerRabinHelper(a, m, k, n))
        {
            return 0;//测试失败，明确是合数
        }
    }
    return 1;
}
```

前面介绍过，米勒–拉宾检验算法是一个概率方法，但是如果加上限制条件，米勒–拉宾算法

也可以作为一种确定性算法使用。限制条件就是数的范围。根据数学家的证明，只要用 2 和 3 作为随机数进行两次检验，就可以 100%正确地检验小于 1373653 的所有素数。再比如，只要用 2、3、5、7、11 作为随机数进行 5 次检验，就可以 100%正确地检验小于 2152302898747 的所有素数。这些经过证明的经验值都是 100%正确的，但是不在上述范围中的其他大素数的判断，目前还只能是概率结果，也就是说，即使能通过检验，也要打上“伪素数”的标签。

18.2 RSA 算法原理

传统的加密模式一般是信息发送者用特定的密钥对信息加密（加密和解密算法都是公开的），然后将密文传递给接收者，同时还要将密钥告诉接收者，这样接收者就可以用密钥对密文进行解密。这种方式的隐患在于密钥的传递，密钥在传递过程中有可能被截取，此外，密钥分发出去以后就很难控制分发的范围，一旦失控，发出去的加密信息就等于是明文了。

1976 年，维特菲尔德・迪菲（Whitfield Diffie）和马丁・赫尔曼（Martin Hellman）在一篇革命性文章“密码学的新方向”（New Directions in Cryptography）一文中提出了一种使用非对称密钥的密码学新方法，可以在不用传递密钥的情况下完成信息的加密和解密。这就是现代非对称公钥体系的基础，在这种体系中，发送者要给接收者传递密文，首先要得到接收者对外发布的公钥，然后用该公钥加密信息，并将加密的信息发送给接受者。接收者受到密文后用自己的私钥对信息解密，在这个过程中，不需要密钥传递，接受者的私钥自己保管，不对外公开。

这种思想也对密码学家提出了新的挑战，也鼓舞了很多数学家寻找一种满足非对称公钥体系的加密算法。第二年，美国麻省理工学院的罗纳德・李维斯特（Ron Rivest）、阿迪・萨莫尔（Adi Shamir）和伦纳德・阿德曼（Leonard Adleman）三位研究员在“实现数字签名和公钥密码体制的一种方法”一文中首次提出了一种非对称公钥加密算法，因为三个人的姓氏首字符分别是 R、S、A，这种算法就被命名为 RSA 算法。经过四十多年的发展，RSA 算法已经成为现代非对称公钥体系中最基本也是目前应用最广泛最有影响力的公钥加密算法。

18.2.1 RSA 算法的数学理论

在研究 RSA 算法的数学原理之前，先来介绍几个数学概念。首先是“同余”，假定三个整数 a、b 和 $n(n \neq 0)$，如果 a 和 b 的差是 n 的整数倍，则称 a 在模 n 时与 b 同余，记做 $a \equiv b(\mathrm{mod}\ n)$。可以将同余简单理解为等式：$a-b=kn$，$k$ 为任意整数。然后是“欧拉函数”，欧拉函数 $\varphi(n)$定义为所有小于或等于 n，且与 n 互素的正整数的个数，$\varphi(n)$的值又被称为 n 的欧拉数。以 8 为例，1、3、5、7 都与 8 互素，所以就有 $\varphi(8)=4$。当 n 是素数时，$\varphi(n)=n-1$，因为所有比 n 小的数都与它互素。欧拉函数还有一个特性，当 n 可以分解为两个互素的数的乘积时 n 的欧拉函数就是两个因子的欧拉函数的乘积，即 $\varphi(n)=\varphi(pq)=\varphi(p)\varphi(q)=(p-1)(q-1)$。最后是“乘法逆元”，若 $ab\equiv 1(\mathrm{mod}\ n)$，则称 b 为 a 在模 n 的乘法逆元，b 可以表示为 a^{-1}。第 17 章已经介绍过，可以使用欧拉算法求解乘法逆元，相关算法的实现在第 17 章已经给出。

RSA 算法基于一个十分简单的数论事实：将两个大素数相乘十分容易，但那时想要对其乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥的一部分。由此可知，密钥的生成是 RSA 算法的核心，先来看看 RSA 密钥对的生成过程。

(1) 任意选择两个大素数，p 和 q，计算出 $n=p\times q$，n 又被称为 RSA 算法的公共模数。

(2) 计算 n 的欧拉数 $\varphi(n)=(p-1)(q-1)$。

(3) 随机选择加密密钥指数，从$[0,\varphi(n)-1]$中选择一个与 $\varphi(n)$互质的数 e 作为公开的加密指数。

(4) 求解与 e 对应的解密指数 d，d 和 e 满足条件：$(d\times e)\equiv 1 \bmod \varphi(n)$。因为 e 和 $\varphi(n)$互素，因此可以利用扩展欧几里得算法求解同余方程，得到唯一整数解 d。

(5) 销毁 p 和 q，妥善保存私有密钥 SK=(d, n)，将公开密钥 PK=(e, n)分发给希望给你发送信息的人。

由以上过程可知，在 p 和 q 不可知的情况下，要得到私有密钥的解密指数 d，必须知道 $\varphi(n)$，而 $\varphi(n)$必须通过第 2 步给出的方法计算，也就是说，必须要分解公共模数 n，重新得到 p 和 q。对 n 的分解是个数学难题，目前没有有效的方法可以快速分解 n，n 越大越难分解，这就是 RSA 算法的数学原理。以目前计算机的处理能力，当 n 大到一定的程度，可以认为是不可分解的，这也是 RSA 算法安全性的基本保证。

18.2.2 加密和解密算法

RSA 算法的加密和解密其实就是模幂运算，这也是我为什么说 RSA 算法简单的原因，因为 18.1 节已经给出了加密和解密的算法实现代码，就是 `ModularPower()`函数。假设 M 是明文，C 是密文，加密的过程就是：

$$C = M^e \pmod n$$

解密的过程就是：

$$M = C^d \pmod n$$

现在举个经典的密码学示例来解释一下 RSA 加密和解密的过程。爱丽丝希望鲍勃给她发送的信息进行加密，她首先选择两个素数 p=11 和 q=13，计算它们的乘积，得到公共模数 n=143，同时计算出 n 的欧拉数 $\varphi(n)$=120。接下来爱丽丝需要选择一个小于 119（$\varphi(n)-1=119$），且与 $\varphi(n)$互素的数作为加密指数 e，爱丽丝选择 e=7。然后爱丽丝需要求解同余方程 $7d\equiv 1 \bmod 120$，得到 d=103。最后，爱丽丝销毁 p 和 q，将(7,143)作为公开密钥发送给鲍勃，将(103,143)作为私钥自己保存。鲍勃需要发送信息 M=85 给爱丽丝，他先用爱丽丝的公钥对 M 进行加密，得到密文 $C=85^7 \pmod{143} = 123$，然后将密文 C=123 发送给爱丽丝，爱丽丝得到 C 后，用私钥对 C 进行解密，得到明文 $M=123^{103} \pmod{143}=85$。

以上就是整个 RSA 加密和解密的过程，其核心就是模幂运算，使用的过程很简单，但是简单并不意味着不安全，接下来要介绍一下 RSA 算法的安全性。

18.2.3 RSA 算法的安全性

人们在提到 RSA 加密的时候，都会附带一个很重要的参数，就是 RSA 密钥长度，比如 1024 比特的 RSA 密钥，2048 比特的 RSA 密钥等。这里的 1024 和 2048 实际上是 RSA 密钥中公共模数 n 的二进制位长度，n 越大就越难分解，这就是通常人们会认为 1024 比特的 RSA 密钥要比 512 比特的 RSA 密钥更安全的原因。但是从数学上看这个问题，对 n 的分解并没有被证明是 NP 问题，也就是说，增加 n 的长度就能提高安全性还没有被用数学的方法证明（当然，也没有被证伪）。当然，RSA 加密的安全性是由很多因素决定的，比如组成 n 的两个随机素数 p 和 q 的选择就很有讲究。p 和 q 必须是随机生成的强素数，绝对不能用别人用过的素数，或者从某个素数表中选择 p 和 q。p 和 q 的差值应该尽量大，增加分解 n 计算的难度。

加密指数 e 的选择也很重要，e 越大计算量就越大，为了加快加密计算的速度，RSA 算法对公钥 e 选择通常是 3、5、17、257 或 65537。X.509 证书体系建议使用 65537，PEM 建议使用 3，PKCS#1 建议使用 3 或 65537。但是使用太小的 e 会引入小指数攻击问题，在原始数据中填充随机数值，使得 $m^e \pmod n \neq m^e$，可以有效地抵抗小指数攻击。因此使用 RSA 加密数据通常都会指定随机值填充模式，单纯的直接用模幂算法加密数据是不安全的。

除此之外，还有针对明文破解的选择密文攻击方式。攻击者知道了 A 的公开密钥(e, n)，同时截获了用 A 的公钥加密的信息 $Y=X^e \pmod n$。攻击者首先选择一个 $r(r<n)$，计算 $Y_1=r^e \pmod n$，这意味着用 A 的私钥对 Y_1 解密可得到 r，即 $r=Y_1^d \pmod n$。接下来，攻击者计算 $Y_2=Y\times Y_1 \pmod n$，求解 r 的模 n 乘法逆元 $t=r^{-1} \pmod n$。因为 $r=Y_1^d \pmod n$，所以 $t=Y_1^{-d} \pmod n$。现在，攻击者以验证身份的名义将 Y_2 发给 A，请 A 对消息 Y_2 签名，于是得到 $S=Y_2^d \pmod n$。最后，攻击者做以下计算：

$$t\times S=(Y_1^{-d}\,Y_2^d)\pmod n$$

将 $Y_2=Y\times Y_1 \pmod n$代入上式得到：

$$t\times S=(Y_1^{-d}\,Y_2^d)\pmod n=(Y_1^{-d}\,Y^d\,Y_1^d)\pmod n=Y^d \pmod n=X$$

最终攻击者在不知道私有密钥 d 的情况下得到了明文 X。

以上选择密文攻击过程关键的一步就是攻击者需要骗取 A 对 Y_2 进行签名，只要用户 A 不对来历不明的数据直接签名，就可以阻断这种攻击。实际上，RSA 的签名是不对数据直接计算的，而是像加密一样要填充一些随机数值，具体的签名算法请看 18.4 节的介绍。

18.3 数据块分组加密

前面我们讨论了 RSA 加密的数学原理和加密解密过程的算法实现，但是所给出的算法实现还都是在数学领域的大数计算，怎么将其应用到现实生活领域呢？加密和解密领域最典型的问题就是对数据分组加密，RSA 同样支持对数据分组加密。和 DES、AES 这样的分组加密算法不同，

RSA 算法所支持的每个数据分组的大小不仅与 RSA 密钥长度有关，还和数据分组的填充模式（padding scheme）有关。填充模式是和 RSA 算法安全性相关的内容，不同的填充方式需要插入原始数据中的 padding 数据量不一样，因此会影响数据分组中有效载荷的大小。

RSA 算法有多种填充模式，常用的填充模式有 PKCS #1 1.5 和 OAEP 两种，我们将在后面介绍这两种填充模式，这里只给出使用两种填充模式对原始数据载荷大小的影响。RSA 加密算法每个数据分组的有效载荷大小与密钥长度和填充模式的关系如表 18-1 所示。

表18-1　RSA填充模式与数据分组大小关系表

填充模式	RSA密钥长度（位）	输入分组有效载荷（字节）	输出分组长度（字节）
PKCS #1 1.5	768	85	96
PKCS #1 1.5	1024	117	128
PKCS #1 1.5	2048	245	256
OAEP	768	54	96
OAEP	1024	86	128
OAEP	2048	214	256

18.3.1　字节流与大整数的转换

被加密的数据可以理解为字节流，对数据加密时，除了按照表 18-1 给出的输入分组有效载荷大小对字节流进行分组外，还要将分组后的数据转换成大整数才能进行 RSA 加密运算。完成加密运算后，还要将计算得到的大整数转换成字节流数据，这样才能进行存入文件或通过网络传送。解密的过程与之类似，都需要一套大整数与字节流互相转换的方法，这样 RSA 算法才具有实用性。

其实将加密数据转换成大数对象的方法非常朴实无华，如果将大数也看成按照顺序在内存中存放的字节流，这种转换就一目了然。只要逐字节将加密数据转换成 `CBigInt` 类的大数位数组（`m_ulValue`），并正确设置大数位的位数（`m_nLength`），即可完成加密数据转换成大数对象的过程。反之亦然，只要将 `CBigInt` 类的大数位数组中的数据逐字节转换到指定的缓冲区，即完成大数对象转换成字节流数据的过程。转换过程唯一需要注意的是对数据中 0 的特殊处理，对于 `CBigInt` 大数对象来说，最高的大数位不能是 0，如果是 0，则要调整 `m_nLength`。

18.3.2　PCKS 与 OAEP 加密填充模式

18.2.3 节介绍 RSA 算法的安全性时，提到为了对抗小指数攻击，需要在原始数据中添加随机填充数据以提高 RSA 的安全性。常用的填充模式有 PKCS #1 1.5 和 OAEP 两种，PKCS 的全称是公钥加密标准（Public-Key Cryptography Standards），是 RSA 实验室发布的一个标准。OAEP 的全称是最优非对称加密填充（Optimal Asymmetric Encryption Padding），是一个比 PKCS #1 1.5 新的填充标准，被 PKCS #1 2.0 接受为新的填充标准。理论上 OAEP 有比 PKCS #1 1.5 更好的安全

性，但是从兼容性来说，PKCS #1 1.5 具有更好的兼容性。

从表 18-1 可以看出，PKCS #1 1.5 需要额外的 11 字节进行随机填充，而 OAEP 需要额外的 42 字节用于随机数据填充，而且 OAEP 的填充方式更具随机化特性，由此可知，OAEP 填充方式对原始数据造成的混乱程度（熵值）比较大，具有更好的安全性。这里的更好是相对的，并不是说 PKCS #1 1.5 填充方式不安全。当你在两个不同的系统之间传递 RSA 加密的信息时，填充模式的兼容性就是需要特别注意的地方，PKCS #1 1.5 因为发布得早，应用更广泛一些，兼容性就更好一些。

前面提到 PKCS #1 1.5 需要额外的 11 字节进行随机数据填充，实际上是不完全正确的。只有当数据的有效载荷足够多的时候，PKCS #1 1.5 填充的长度才被压缩为 11 字节，当有效载荷不足的时候，实际需要填充的数据会超过 11 字节。总的来说，PKCS #1 1.5 填充可以分为四部分，分别是一字节前导 0，一字节标志 T 和 $k-|M|-3$ 字节的随机数据和一字节的截断符号 0。k 是 RSA 密钥公共模数的字节长度，$|M|$表示实际载荷数据长度，$|M|$通常要满足$|M|\leqslant k-11$，当$|M|=k-11$ 的时候，PKCS #1 1.5 要求的最小填充长度是 11 字节时，其结构如图 18-1 所示：

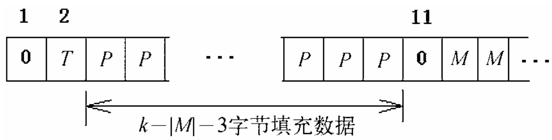


图 18-1 PKCS #1 1.5 填充模式

其中标志字节 T 表示拼凑填充数据的方法，如果 T 是 0，表示填充数据 P 全部是 0；如果 T 是 1，表示填充数据 P 全部是 0xFF；如果 T 是 2，表示填充数据是 $k-|M|-3$ 个 1–0xFF 之间的随机数据。前导位设为 0 是为了保证转换后得到的大数是正数，这 $k-|M|$字节的填充数据放在实际的数据载荷之前，构成完整的加密数据分组，然后转换成大整数进行加密计算。对加密过的数据解密时，解密计算后得到的数据也是包含填充数据的，要从中提取出有效数据载荷，其方法就是从前导 0 和标志 T 开始向后搜索，直到找到截断 0 标志为止（随机填充数据不会是 0），在这之间的都是填充数据，剩下的就是有效数据载荷。

相对于 PKCS #1 1.5 来说，OAEP 稍显复杂一点。OAEP 是 Mihir Bellare 和 Philip Rogaway 两位密码学家提出的一种加密方案，后来被 PKCS #1 2.0 接受为标准，因此也被称为 PKCS #1 2.0 填充方案。OAEP 模式中明文载荷的长度$|M|$要满足$|M|\leqslant k-2\text{hLen}-2$，其中 hLen 是 OAEP 模式所选择的哈希函数的输出长度。OAEP 模式的哈希函数和掩码生成函数都不是固定的，可以通过参数化配置和选择，如果选择 SHA（Secure Hash Algorithm），输出长度是 20 字节，则明文载荷不能超过 $k-42$ 字节。OAEP 模式需要指定一个与明文有关联的标签 L，如果没有指定，则默认 L 是空。OAEP 的加密过程如下。

(1) 计算标签 L 的哈希输出，得到一个长度为 hLen 的字节串 lHASH=HASH(L)；

(2) 生成一字节串 PS，内容是 0，长度为 $k-|M|-2\text{hLen}-2$ 字节。当$|M|=k-2\text{hLen}-2$ 时，

PS 长度有可能是 0；

(3) 连接 IHASH，PS，一字节的 0x01 和明文 *M*，得到一个 $k - \text{hLen} - 1$ 字节长度的字节流 DB= IHash|PS|0x01|*M*；

(4) 生成一个长度为 hLen 的随机字节串 seed，并使用掩码生成函数将其转换为长度 k–hLen–1 字节的掩码 DBmask，DBmask= MGF(seed, k – hLen – 1)；

(5) 用 DBmask 与 DB 做掩码计算得到 mDB=DB ⊕ DBmask；

(6) 用掩码生成函数将 mDB 转换为长度为 hLen 字节的掩码 seedMask，将其与随机字节串 seed 做掩码计算，得到 Mseed。即 seedMask=MGF(mDB, hLen)，mSEED= seed ⊕ seedMask；

(7) 将一字节的 0x00、mSEED 和 mDB 拼在一起，组成长度为 k 的加密数据分组 EM，即 EM=0x00|mSEED|mDB，然后将 EM 转换为大整数即可进行 RSA 加密计算。

OAEP 解密的过程与加密过程相反，首先要将加密数据转换成大整数，进行 RSA 解密计算，然后将解密后的大整数转换成字节流，此时就得到加密过程第 7 步拼接的数据分组 EM，然后再按照以下过程分解出原始加密数据 *M*。

(1) 计算标签 *L* 的哈希输出，得到一个长度为 hLen 的字节串 IHASH=HASH(*L*)；

(2) 从 EM 中分解出 mSEED 和 mDB，用掩码生成函数将 mDB 转换为长度为 hLen 字节的掩码 seedMask，即 seedMask=MGF(mDB, hLen)；

(3) 计算 seed=mSEED ⊕ seedMask，然后用掩码生成函数将其转换成长度为 k–hLen–1 字节的掩码 DBmask，DBmask= MGF(seed, k – hLen – 1)；

(4) 根据 DBmask 和 mDB 计算得到 DB，DB=mDB ⊕ DBmask，如果解密计算过程没有错误，DB 的内容应该和加密过程第(3)步得到的 DB 是一样的；

(5) 分解 DB，首先匹配一下 IHash 与第 1 步算出来的是否一致，如果不一致说明解密错误。如果一致，则匹配一连串 0 和一个 0x01，如果能匹配，则剩下的就是原始明文 *M*，如果不能匹配，说明解密错误。

18.3.3　数据加密算法实现

分组数据的加密过程，实际上是一个和填充模式捆绑在一起对数据进行处理的过程。现在就以简单的 PKCS #1 1.5 填充模式为例，说明一下 RSA 分组数据加密过程。这个过程正如 `Rsa_Pkcs15_Encrypt_Block()`函数所展示的那样，首先是填充前导 0，然后是 *T* 标识，我们选择 *T*=2，这也是 PKCS #1 1.5 推荐的模式。接下来是随机字节串，长度由 $k - |M| - 3$ 计算得到，`GeneratePkcsPad()`函数产生长度为 pad_len 的随机字节串。在拼接原始数据之前，还要再添加一个截断标识 0。完成填充之后，将数据转化成大整数，用模幂运算进行加密，得到密文大数 *c*，将 *c* 转换成字节串，即可作为加密后的数据进行存储或分发。

```
int Rsa_Pkcs15_Encrypt_Block(CBigInt& e, CBigInt& n, int kbits,
    void *pSrcBlock, int srcSize, CBigInt& c)
{
    int k = kbits / 8;
```

```
    int pad_len = k - srcSize - 3;

    char *padBlock = new char[k];
    if(padBlock == NULL)
        return -1;

    padBlock[0] = 0x00;
    padBlock[1] = 0x02; //填充随机数
    GeneratePkcsPad(2, padBlock + 2, pad_len);
    padBlock[pad_len + 2] = 0x00;
    memcpy(padBlock + k - srcSize, pSrcBlock, srcSize);
    CBigInt em;
    em.GetFromData(padBlock, k);//OS2IP
    c = ModularPower(em, e, n);

    delete[] padBlock;

    return k;
}
```

18.3.4 数据解密算法实现

分组数据的解密过程也是和填充模式捆绑在一起的处理过程。首先将得到的密文转换成大整数 *c*，然后对 *c* 进行模幂运算解密，得到密文 *em*，最后将 *em* 转换成字节串，并分离出随机填充信息，得到原始明文信息。Rsa_Pkcs15_Decrypt_Block()函数展示的就是分组解密的实现过程，其中分离填充信息需要用到截断标识，就是插入到随机填充字节串和原始数据之间的那个 0。PKCS #1 1.5 要求填充的随机字节都是 1~0xFF 的数值，因此，从填充标识 *T* 开始搜索，遇到的第一个 0 肯定就是截断标识，其后跟的就是原始数据。

```
int Rsa_Pkcs15_Decrypt_Block(CBigInt& d, CBigInt& n, int kbits,
    CBigInt& c, void *pDecBlock, int blockSize)
{
    char *padBlock = new char[kbits / 8];
    if(padBlock == NULL)
        return -1;

    CBigInt em = ModularPower(c, d, n);
    int dataSize = em.PutToData(padBlock, kbits / 8);
    int pad_len = 2;
    for(int i = 2; i < dataSize; i++)
    {
        pad_len++;
        if(padBlock[i] == 0)
            break;
    }
    memcpy(pDecBlock, padBlock + pad_len, dataSize - pad_len);
    delete[] padBlock;

    return dataSize - pad_len;
}
```

18.4 RSA 签名与身份验证

RSA 密钥中的加密指数 e 和解密指数 d 是关于模 $\varphi(n)$ 的乘法逆元，这个关系使得 RSA 的加密和解密过程具有一个很有意思的特点。这个特点就是用 e 加密的数据可以用 d 解密，反过来，用 d 加密的数据也可以用 e 解密。利用这个特点，RSA 算法还可以用来做数字签名和身份验证。数字签名是只有信息发送者才能产生、别人无法伪造的一段信息，这段信息还可以用来验证发送者所发送信息的真实性。可以这样理解，数字签名就是信息发送者用自己的私钥对一段公开信息加密后得到密文信息，因此它具有两个特征：

- 任何人都可以利用发送者的公钥验证签名的有效性（对其解密并比较）
- 签名具有不可伪造性和不可否认性（因为只有发送者有与公钥对应的私钥）

签名的验证就是利用发送者的公钥对加密信息解密，然后比较解密后的信息是否是原始信息。数字签名的一般过程是：用户 A 用选择的哈希算法计算出文件 M 的 HASH 值，然后用自己的私钥对这个 HASH 值进行加密，这个过程就是“签名”。现在 A 把文件 M（不加密）和加密后的 HASH 值发给 B，B 于是用 A 的公钥对 HASH 值解密，然后再计算文件 M 的 HASH 值，比较文件的 HASH 值是否和 A 发来的 HASH 值一样，如果一样则说明文件确实是 A 发来的，并且文件没有被修改。否则就说明文件不是 A 发出的，或者文件在传递过程中被篡改了。由此可知，数字签名除了证明文件 M 是 A 发出的，还可以验证文件 M 是否被其他人（包括接受者）篡改或伪造。

身份验证的过程和签名的过程类似，当 B 需要验证 A 的身份时，就将一段信息发给 A，请 A 对其进行签名（加密）。得到 A 的签名后，B 使用 A 的公钥对签名解密，验证是否和自己发给 A 的信息一致，以此验证 A 身份。其他人没有 A 的私钥，无法伪造 A 的签名。

和 RSA 的加密一样，使用 RSA 签名一样需要面对各种攻击，因此，不要随便对某一个人发过来的东西进行签名（有潜在危险）。如果必须要这么做（比如为了验证身份），最好先用哈希算法计算出信息的 HASH 值，然后对 HASH 值进行签名。

18.4.1 RSASSA-PKCS 与 RSASSA-PSS 签名填充模式

RSA 的签名和身份验证过程，面临着和 RSA 加密过程一样的攻击方法，因此 RSA 实验室对签名算法也制定了和加密过程一样的随机填充模式标准——*带填充的签名算法*。PKCS 制定了两种带填充的签名算法，分别是 RSASSA-PSS 和 RSSSA-PKCS #1 1.5。尽管从理论上讲 RSASSA-PSS 比 RSSSA-PKCS #1 1.5 有更好的健壮性，但是目前还没有发现针对 RSSSA-PKCS #1 1.5 的有效的攻击手段。RSSSA-PKCS #1 1.5 的签名算法已经在很多系统上得到了广泛的应用，具有很好的兼容性，不过 PKCS 标准建议新的应用系统应该能够平滑地过度到 RSASSA-PSS 算法。

RSSSA-PKCS #1 1.5 签名算法的填充方式和 PKCS #1 1.5 加密算法的填充方式类似，主要由以下六部分组成：

$$EM = 0x00 \mid 0x01 \mid PS \mid 0x00 \mid T \mid H$$

首先是前导 0，*T* 标识固定是 1，也就是说，PS 使用随机长度的 0xFF 填充，跟在截断标识 0 后的 *T* 是一个与哈希算法相关的字节串，其内容与哈希算法有关，但是每种哈希算法对应的 *T* 的内容和长度是固定的。哈希算法与 *T* 的关系如表 18-2 所示。

表18-2 PKCS #1 1.5签名算法中哈希算法与*T*填充内容的关系

Hash	*T*
MD5	30 20 30 0c 06 08 2a 86 48 86 f7 0d 02 05 05 00 04 10
SHA-1	30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14
SHA-224	30 2d 30 0d 06 09 60 86 48 01 65 03 04 02 04 05 00 04 1c
SHA-256	30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20
SHA-384	30 41 30 0d 06 09 60 86 48 01 65 03 04 02 02 05 00 04 30
SHA-512	30 51 30 0d 06 09 60 86 48 01 65 03 04 02 03 05 00 04 40

H 是明文 *M* 的哈希值，填充数据 PS 的长度等于 $k - |T| - |H| - 3$，其中 k 是 RSA 密钥公共模数 n 的字节长度，$|T|$是填充 *T* 的长度，$|H|$是明文 *M* 的哈希值的长度。完成填充后得到 EM，将其转换成大整数，用私钥进行加密，就得到数字签名。

RSSSA-PKCS #1 1.5 的签名验证过程与解密过程类似，首先将签名数据转换成大整数，然后用公钥解密，就得到签名之前的填充状态 EM。分解 EM，得到签名中的原始信息的哈希值 *H*，然后将这个 *H* 与原始明文计算出来的哈希值比较，如果一致则签名验证成功，如果不一致，说明这是一个无效的签名，或者是伪造的签名。

18

RSSSA-PSS 签名填充模式是一种新的签名填充，在 RSSSA-PKCS #1 v2.1 中被接受为 PKCS 的标准。RSSSA-PSS 源于 Mihir Bellare 和 Philip rogaway 发明的概率填充方案（Probabilistic Signature Scheme），将其应用于 RSA 加密体系，就是 RSSSA-PSS。RSSSA-PSS 签名算法的输入参数是明文 *M* 和签名体最大比特长度 emBits，emBits 的最小值不能小于 8hLen+8sLen+9，其签名过程如下。

(1) 计算明文 *M* 的哈希值 mHash=HASH(*M*)，长度为 hLen，生成一个随机字节串 salt，长度为 sLen；

(2) 拼接 MP=0 0 0 0 0 0 0 0 |mHash|salt，计算 MP 的哈希值 *H*=HASH(MP)，*H* 长度是 hLen；

(3) 生成由 0 字节组成的字节串 PS，长度为 emLen–hLen–sLen – 2，emLen=$\lceil emBits/8 \rceil$。PS 的长度也可以是 0；

(4) 令 DB = PS|0x01|salt，DB 是一个长度为 emLen – hLen – 1 的字节串；

(5) 用掩码生成函数将 MP 的哈希值 *H* 转换成长度为 emLen – hLen – 1 的掩码，DBmask= MGF(*H*, emLen – hLen – 1)；

(6) 计算 mDB = DB ⊕ DBMask，将 mDB 的左边最高有效位的 8emLen–emBits 个比特置为 0；

(7) 拼接 mDB、MP 的哈希值 *H* 以及一个字节的固定值 0xBC，得到 EM=mDB|*H*|0xBC。将 EM 转换成大整数，用私钥加密即可得到 RSSSA-PSS 填充的数字签名。

RSSSA-PSS 签名的验证过程首先要将签名体转换成大整数，用公钥解密得到 EM，然后按照以下步骤验证 EM。

(1) 从左到右对 EM 进行分解，前 emLen–hLen–1 个字节是 mDB，接下来是 hLen 字节的 *H*，最右边是一个字节的固定值 0xBC，如果最右边一个字节不是 0xBC，则输出“验证失败”，并停止；

(2) 如果 mDB 左边最高 8emLen – emBits 个比特不是 0，则输出“验证失败”，并停止；

(3) 用掩码生成函数将 *H* 转化成 emLen – hLen – 1 字节的掩码 DBmask= MGF(*H*, emLen – hLen – 1)，并计算出 DB=mDB ⊕ DBMask；

(4) 将 DB 的左边最高有效位的 8emLen – emBits 个比特置为 0，并判断 emLen – hLen – sLen – 1 位置的一个字节是否是 0x01，如果不满足这两个条件，则输出“验证失败”，并停止；

(5) DB 的最后 sLen 个字节是 salt，计算出明文 *M* 的哈希值 mHask，并拼接出 MP=0 0 0 0 0 0 0 0 |mHash|salt；

(6) 计算 MP 的哈希值，并与 *H* 比较，如果相等则输出“验证成功”，否则输出“验证失败”，并停止。

由此可见，RSSSA-PSS 和其他签名填充模式一样，也是遵循“先哈希再签名”的原则，不直接使用明文 *M* 签名。

18.4.2　签名算法实现

有了上一节分析的签名算法的原理和实现步骤，写出签名算法的实现代码就易如反掌。Rsa_Pkcs15_Sign()函数就是 RSSSA-PKCS 算法的实现，采用了 MD5 作为哈希值计算函数，可以看到其填充方式和 PKCS 加密填充方式类似，最终 pSignBuf 得到 *k*bits（*k* 字节）的签名数据：

```
int Rsa_Pkcs15_Sign(CBigInt& d, CBigInt& n, int kbits,
    void *pSrcData, int dataSize, void *pSignBuf, int bufSize)
{
    int k = kbits / 8;

    char *padBlock = new char[k];
    if(padBlock == NULL)
        return -1;

    unsigned char md5Hash[MD5_DIGEST_SIZE] = { 0 };
    CalcMD5Hash(pSrcData, dataSize, md5Hash);

    int pad_len = k - MD5_DIGEST_SIZE - Md5SignPadSize - 3;
    padBlock[0] = 0x00;
    padBlock[1] = 0x01; //填充全 xFF
    GeneratePkcsPad(1, padBlock + 2, pad_len);
    padBlock[pad_len + 2] = 0x00;
    memcpy(padBlock + pad_len + 3, Md5SignPadding, Md5SignPadSize);
    memcpy(padBlock + pad_len + 3 + Md5SignPadSize, md5Hash, MD5_DIGEST_SIZE);
    CBigInt em;
    em.GetFromData(padBlock, k);//OS2IP
    CBigInt  c = ModularPower(em, d, n);
```

```
    c.PutToData((char *)pSignBuf, k);

    delete[] padBlock;
    return k;
}
```

18.4.3 验证签名算法实现

RSSSA-PKCS 算法的签名验证实现也不复杂，出于篇幅考虑，Rsa_Pkcs15_Verify()函数给出了概念性实现代码，这个函数只处理了具体哈希值的解析和判断（这已经是签名验证算法的主体），没有对填充数据的格式校验，有兴趣的读者可自行加上校验，使之成为更有实用性的签名验证算法。

```
bool Rsa_Pkcs15_Verify(CBigInt& e, CBigInt& n, int kbits,
    void *pSignData, int dataSize, void *pSrcData, int srcSize)
{
    char *padBlock = new char[kbits / 8];
    if(padBlock == NULL)
        return false;

    CBigInt c;
    c.GetFromData((const char *)pSignData, dataSize);
    CBigInt em = ModularPower(c, e, n);
    int emSize = em.PutToData(padBlock, kbits / 8);
    int pad_len = 2;
    for(int i = 2; i < emSize; i++)
    {
        pad_len++;
        if(padBlock[i] == 0)
            break;
    }

    unsigned char md5Hash[MD5_DIGEST_SIZE] = { 0 };
    CalcMD5Hash(pSrcData, srcSize, md5Hash);

    int result = memcmp(padBlock + pad_len + Md5SignPadSize, md5Hash, MD5_DIGEST_SIZE);

    delete[] padBlock;

    return (result == 0);
}
```

18

18.5 总结

这一章我们介绍了 RSA 算法的原理，把看似神秘的 RSA 算法在放大镜下看了个底朝天，原来如此嘛，你应该有这种感觉吧？蒙哥马利算法、米勒–拉宾算法、欧几里得算法，一个个看似高高在上的名词，原来有如此平易近人的实现，举重若轻，算法的乐趣尽在于此吧。现在，你可以用本章的算法给自己弄个签名什么的，也可以把自己的公钥散发出去，让别人也给你发送加密邮件，最重要的，这些都是你自己实现的。

还记得介绍广义黎曼猜想时提到的英国数学家哈代吗？有一次，哈代要乘船渡北海回英国，那天天气恶劣，浪涛汹涌，而船又很小，因此他在船开之前就写了一张明信片寄给丹麦物理学家波尔（Harald Bohr），上面只写了一句话："我已经证明了黎曼猜想。哈代。"哈代寄这张明信片的用意是：万一这船沉入大海，哈代死了，世人就会认为哈代真的解决了这个世界数学难题，而为这个解法及哈代一起沉入海底而惋惜。但是上帝如此不喜欢哈代，一定不会让哈代享有解决这个著名难题的声誉，肯定不会让这艘船沉入大海，于是哈代就可以平安回到英国，这样这张明信片就是他的护身符了。本章有些内容还是比较严肃的，大家看看这个乐一下吧，顺便说一下，这不是笑话，是真事儿。

18.6　参考资料

[1] Montgomery P. *Modular Multiplication Without Trial Division*. Math. Computation, Vol. 44:519-521, 1985

[2] Schneier B. 应用密码学：协议、算法与 C 源程序（中文版）. 吴世忠，祝世雄，张文政，等译. 北京：机械工业出版社，2004

[3] 王小云，王明强，孟宪萌. 公钥密码学的数学基础. 北京：科学技术出版社，2013

[4] Stinson D R. 密码学原理（第 3 版）. 北京：电子工业出版社，2009

[5] 乔纳森・卡茨，耶胡达・林德尔. 现代密码学:原理与协议. 北京：国防工业出版社，2011

[6] Bajard J-C, Didier L-S, Komerup P. *An RNS Montgomery Modular Multiplication Algorithm*. *IEEE Transaction on Computers*, 47(7):766–776, July 1998

[7] Koc C K, Acar T, Burton S, et al. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3): 26–33, June 1996

[8] Quisquater J-J, Couvreur C. Fast Decipherment Algorithm for RSA Public-Key Cryptosystem. *Electronics Letters*, Vol. 18, pp. 905–907, October 1982

[9] Wu C-H, Hong J-H, Wu C-W. VLSI Design of RSA Cryptosystem Based on the Chinese Remainder Theorem. *Journal of Information Science and Engineering* 17, 967-980, 2001

[10] Rivest RL, Shamir A, Adleman L. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. *Communications of the ACM*, 21(2): 120-126, February 1978

[11] RSA Laboratories. *PKCS #1 v2.1: RSA Encryption Standard*, June 2002

第 19 章 数独游戏

数独游戏（SUDOKU）是一种数学智力拼图游戏，起源于 18 世纪末的瑞士，当时的瑞士数学家莱昂哈德·欧勒发明了“拉丁方块”游戏，但并没有受到人们的重视。直到 20 世纪 70 年代，美国杂志才以“数字拼图”（number place puzzles）游戏的名称将它重新推出，结果风靡一时。日本随后接受并推广了这种游戏，并且将它改名为“数独”，大致的意思是“独个的数字”或“只出现一次的数字”。数独游戏在日本非常流行，许多报纸和杂志都会刊登数独游戏。在日本的地铁上经常看到手拿报纸和铅笔、眉头紧锁的人，那就是在玩数独游戏。玩数独游戏不需要学习额外的知识，也不像字谜游戏那样需要很大的词汇量，大人和小孩都适合玩数独游戏。

数独游戏在流行的过程中产生了很多变形数独，比如格子数演变成 6×6，12×12，甚至是 16×16，还有的规则要求对角线上的数字也要满足不重复的要求。不过，总的来说，这些变种都没有偏离这个游戏的基本规则。本章就介绍一下传统的 9×9 格子的数独游戏（九宫数独），并给出一种以候选数法为基础的求解数独游戏的算法实现。

19.1 数独游戏的规则与技巧

数独游戏有着独特的规则和技巧，在数独游戏的推广和传播过程中，出现了很多新的规则和新的形式。本节将简单介绍一下数独游戏的基本规则，也是被最广泛接受的规则，当然，也包括在此规则基础上的一些常用技巧。

19.1.1 数独游戏的规则

9×9 格子数独游戏的形式如图 19-1 所示，为了方便描述，一般用大写字母 A ~ I 来标识行，用数字 1 ~ 9 来标识列，这样每个小单元格就有了坐标。数独游戏的规则非常简单，就是在 $9 \times 9 = 81$ 个单元格中填入数字 1 ~ 9。这 81 个单元格又组成 $3 \times 3 = 9$ 个小九宫格，要求填入的数字在每行和每列都不能有重复，同时在每个小九宫格中也不能有重复。游戏开始时会将一些位置上的数字固定下来，这称为提示数（或起始数），根据提示数的位置和数量可以将数独游戏分成不同的难度级别。

	1	2	3	4	5	6	7	8	9
A						6			1
B	9						3	7	6
C	7	1			4				
D	1	7		8					3
E		3						1	
F	6					3		5	8
G					3			6	5
H	3	5	1						2
I	8			1					

图 19-1　一个数独游戏的例子

19.1.2　数独游戏的常用技巧

解决数独问题的技巧大致可分为两类，一类是直观法，另一类是候选数法。直观法就是不借助任何辅助工具，直接利用数独游戏的规则进行求解的方法，一般只需一只铅笔就可以直接在报纸或杂志上玩了。这种方法适合求解简单的数独游戏，对于常见于报纸和杂志上的数独题目（专业数独杂志除外），都可以轻松应对，也是最能体验数独乐趣的一种方法。直观法常用的技巧包括唯一解法、基础排除法、区块排除法、唯余解法、矩形排除法、单元排除法等。这些技巧中唯一解法、基础排除法和唯余解法是基本的技巧，一般简单的数独用这三种解答技巧就可以应付。除此之外，其他几种技巧都对应一种或多种稍微复杂的数独局面，当求解数独过程中遇到与之相似的局面时，应用对应的方法可以起到事半功倍的效果。

候选数法首先要根据数独题目的需要为每个没有确定的单元格建立一个候选数列表，然后根据各种排除方法，逐步排除每个单元格中不可能出现的候选数，当某个单元格对应的候选数列表中只剩下唯一候选数时，这个剩下候选数就是该单元格要填的正确数字。候选数法需要一个准备过程，要为每个单元格建立候选数列表，通常在解题过程中，都是先利用直观法进行求解，直到直观法无法继续时，才使用候选数法。候选数法需要做一些简单的记录来维护候选数列表，因此没有直观法那么直接，但是候选数法适合解决较为复杂的数独难题（比如有多个解的数独问题）。候选数法常用的技巧包括唯一候选数法、隐性唯一候选数法、区块删减法、数对删减法、隐性数对删减法、三链数删减法、隐性三链数删减法、矩形顶点删减法、三链列删减法、关键数删减法、关连数删减法等。

19.2　计算机求解数独问题

用计算机求解数独问题最典型的方法就是使用穷举的方法遍历整个解空间，遍历过程中结合数独规则设置适当的“剪枝”条件排除掉一些明显错误的分支，加快遍历的速度。这种方法的算法实现简单，对于 9×9 的数独题目来说，解题的速度还可以；对于有多个解的高难度数独问题，

也可以轻松应对。这种方法的问题是遍历和回溯都是在最大深度上进行的，对于一个已经固定了28个数字的数独题目来说，遍历和回溯的深度就是81–28=53层，效率很低。另外，为了能利用“剪枝”加快遍历速度，在遍历过程中要频繁地判断当前数独状态是否符合数独规则，而这些判断在很多情况下都是不必要的。

最简单的穷举算法是对每个单元格都进行深度尝试，也就是用1～9分别进行试数，然后利用数独规则对试数进行检查，如果合法则继续对下一个单元格试数，直到所有单元格都填了数字，且检查符合数独规则就算找到一个解。有一些文献或资料提出了一种改进的方法，就是仿照候选数法为每个单元格建立候选数列表，穷举的时候只利用候选数列表中的数字进行试数。这种改进减少了一些明显不正确的尝试。不仅如此，候选数一般是该单元格上可能的合法数字，直接使用候选数进行试数可以避免不必要的有效性检查，这些都能有效地提高算法效率。但是这种方法只是机械地利用候选数列表避免了不必要的尝试，并没有充分利用数独游戏规则对每个单元格的候选数列表进行动态维护，并没有最大限度地发挥候选数列表的作用。

本章要介绍的算法也是一种基于候选数方法的穷举算法，但是本方法不仅利用候选数列表减少不必要的试数次数，还在每次试数的过程中动态维护相关单元格的候选数列表。动态维护候选数列表的优点是不仅可以更快地排除候选数，尽可能多地减少试数的次数，还可以在一次试数的过程中确定多个单元格的数字。在介绍这种方法之前，先来了解一个概念：相关20格。数独中每一个单元格所在的行、列和小九宫格中的20个单元格被称为这个单元格的相关20格。在数独游戏中，在一个单元格填入一个确定数字，则这个单元格的相关20格都会受到影响。对于候选数法来说，这种影响就是候选数的排除。实际上，在对每个单元格试数的过程中，相关20格的候选数列表是会发生变化的，有可能会导致某些单元格出现唯一候选数，还可以利用这一点在一次试数过程中确定多个单元格的数字，减少回溯的次数。另外，在试数的过程中，各个单元格的候选数列表是动态维护的，也就是说这些候选数都是可以填入单元格的有效候选数，因此，整个穷举过程都不需要进行数独规则的合法性检查。当最后被确定的单元格达到81个时，得到的就是一个合法的数独结果。

现在来总结一下本章介绍的算法的要点。首先为每个单元格建立候选数列表，并且对每个已经给出的提示数使用基本排除法，排除与这些提示数有关的相关20格的无效候选数。然后利用枚举的方法对每个还没有确定的单元格进行试数，每进行一次试数，就对这个单元格的相关20格的候选数列表进行排除法维护，如果相关20格中的某个单元格位置符合唯一候选数条件，则将这个单元格设置为确定状态，同时对这个单元格的相关20格的候选数再进行排除（这个过程可能是递归过程）。重复上述过程，直到本次试数结束。当最后一个未确定的单元格完成试数时，得到的结果就是正确的结果，不需要再做数独规则合法检查。

穷举算法的效率主要由穷举所需要搜索的解空间大小所决定，需要搜索的解空间越大，找到正确结果的效率就越低。对于具体的实现算法来说，穷举算法每次深度搜索需要回溯的次数直接决定了解空间的大小，减少回溯次数就能有效地减少穷举需要搜索的解空间大小。本章给出的算法从理论上将可以有效地减少穷举回溯的次数，实际效果如何呢？接下来就给出算法实现，就递

归回溯的深度进行一下验证。

19.2.1 建立问题的数学模型

本书前面介绍过，用计算机求解现实问题，需要解决三个关键问题：计算机求解数学模型的建立、人类语言描述的问题与数学模型的转换和算法设计。对于数独游戏这样简单的问题，建立数学模型可以简化为一系列简单数据结构的定义。这些数据结构不仅要能在计算机系统内表达原始问题，还要有利于设计算法。算法设计可以理解为对表达在定义好的数据结构上的数据的一系列操作和转换，因此，数据结构的定义还要能够对这些操作和转换提供尽可能的便利。

首先，我们需要一个模型表达数独游戏的每个小单元格。根据对题目的理解，每个小单元格可能有两种状态，分别是确定数字的状态和不确定数字的状态。当单元格处于确定的状态时，需要一个属性描述这个确定的数字。当单元格处于不确定的状态时，需要一个属性描述候选数列表。在定义数据结构时，我们当然可以为单元格定义两种数据结构，但是考虑到在数独问题的求解过程中，单元格的状态会发生变化，为了使数据的表达形式统一，便于数独整体数据结构的定义，我们考虑将单元格的数据结构统一定义如下：

```
typedef struct
{
    int num;
    bool fixed;
    std::set<int> candidators;
}SUDOKU_CELL;
```

其中额外附加的 fixed 标志用于标识单元格的两种状态。候选函数列表使用了 STL 的 set 容器，因为在算法实现过程中，对候选数列表最常见的操作就是删除某个候选数，STL 的 set 容器无疑为这个操作提供了最方便的接口。

接下来考虑一下数独整体数据结构的定义。因为我们的算法不需要复杂的规则检查，因此不需要额外的标识，只需要一个二维矩阵表示 81 个单元格，另外再加上一个当前已经确定的单元格计数器即可：

```
typedef struct
{
    SUDOKU_CELL cells[SKD_ROW_LIMIT][SKD_COL_LIMIT];
    int fixedCount;
}SUDOKU_GAME;
```

在算法穷举解空间的过程中，每确定一个单元格的数字，fixedCount 计数器就+1，当 fixedCount 计算器等于 81 时，就表示找到了一个合法的解。

从数据结构定义就可以看出来本算法的数学模型相当简单，但是简单并不意味着粗糙，应用这个模型，将原始数据转换成这个模型以及输出最终的结果都变得非常简单。

19.2.2 算法实现

算法实现的基本思想是从第一个单元格开始搜索，跳过已经确定的单元格，直到找到一个未确定的单元格，然后使用这个单元格的候选数列表对这个单元格开始试数。每试一个数，就将相关 20 格的候选数进行一次排除，如果没有冲突，就从这个单元格之后的一个位置开始继续搜索未确定的单元格，算法整体结构如下所示：

```
void FindSudokuSolution(SUDOKU_GAME *game, int sp)
{
    if(game->fixedCount == SKD_CELL_COUNT)
    {
        std::cout << "Find result :" << std::endl;
        PrintSudokuGame(game);
        return;
    }

    sp = SkipFixedCell(game, sp); //跳过确定单元格
    if(sp >= SKD_CELL_COUNT)
        return;

    int row = sp / SKD_COL_LIMIT;
    int col = sp % SKD_COL_LIMIT;
    SUDOKU_CELL *curCell = &game->cells[row][col];
    SUDOKU_GAME new_state;
    std::set<int>::iterator it = curCell->candidators.begin();
    while(it != curCell->candidators.end())
    {
        CopyGameState(game, &new_state);
        if(SetCandidatorTofixed(&new_state, row, col, *it))
        {
            //试数成功，没有冲突，从后面一个单元格继续
            FindSudokuSolution(&new_state, sp + 1);
        }
        ++it;
    }
}
```

之所以要复制一个新状态，是因为试数过程会改变某些单元格的状态（候选数列表），为了使深度遍历能够正确回溯，需要保持原状态不变。因此，复制一个新状态，并对新状态进行变换是最简单的方法。本算法如果还有提升效率的余地的话，这个状态的处理应该是一个可以改进的地方。

SetCandidatorTofixed()函数的处理是本算法区别于其他算法的核心。对相关 20 格的处理就在这个函数中：

```
bool SetCandidatorTofixed(SUDOKU_GAME *game, int row, int col, int num)
{
    SwitchCellToFixed(game, row, col, num);

    if(!ExclusiveCorrelativeCandidators(game, row, col, num))
        return false;
```

```
        if(!ProcessSinglesCandidature(game, row, col, num))
            return false;

        //到这里说明在[row][col]位置填入 num 没有问题，可以确定这个单元格的数字
        game->fixedCount++;
        return true;
    }
```

SwitchCellToFixed()函数将当前单元格的状态从未确定转换成确定，ExclusiveCorrelativeCandidators()函数按照数独规则从相关 20 格的候选数列表中删除 num。如果此次试数 num 选择不合法，会导致相关 20 格的候选数列表出现空列表的情况，如果出现这种情况，ExclusiveCorrelativeCandidators()函数返回 false 终止试数。ProcessSinglesCandidature()函数对排除候选数后的相关 20 格的候选数列表进行检查，看看是否有唯一候选数的情况，如果某个单元格符合唯一候选数条件，则将此单元格设置为确定状态，同时对这个单元格的相关 20 格再进行排除候选数操作。ProcessSinglesCandidature()函数分别对[row][col]位置所在的行、列和小九宫格进行唯一候选数检查，以行检查的代码为例：

```
        //处理行
        for(int i = 0; i < SKD_COL_LIMIT; i++)
        {
            if(!game->cells[row][i].fixed
                && (game->cells[row][i].candidators.size() == 1))
            {
                int num = *(game->cells[row][i].candidators.begin());
                if(!SetCandidatorTofixed(game, row, i, num))
                    return false;
            }
        }
```

读者可能已经注意到了，这是个递归操作，也就是说，SetCandidatorTofixed()函数如果返回 false，将导致递归回溯到最初的试数单元格位置，此时 FindSudokuSolution()函数的 while 循环内会继续尝试下一个候选数，这正是我们想要的结果。ProcessSinglesCandidature()函数中对列和小九宫格的处理与行处理方法类似，都隐式地包含递归过程。

19.2.3　与传统穷举方法的结果对比

如果用最简单的穷举方法，对于有 26 个提示数的数独游戏，穷举需要搜索的递归深度是 81−26=55 级。我用 1000 个有 26 个提示数的数独游戏进行了对比测试，采用本章给出的算法，穷举需要搜索的递归深度平均在 9 级以内，最多不超过 13 级递归搜索。其中 30%左右的数独游戏基本上在 2 ~ 3 级深度搜索即可找到正确结果。对于芬兰数学家因卡拉给出的号称世界最难的数独游戏，本算法的递归深度也只有 16 级。

19.3　关于数独的趣味话题

芬兰都柏林大学学院的 Gary McGuire 曾经公开了一个证明方法，证明了一个数独具有唯一

解的条件是提示数不能少于 17 个。也就是说，16 个或更少提示数的数独游戏可能会存在多个解。无独有偶，来自澳大利亚自佩斯市西澳大利亚大学的数学家 Gordon Royle 也用另一种方法证明了数独具有唯一解的条件，结论也是 17 个提示数是最低要求。目前，数学家们普遍认可了 McGuire 的证明方法，认为其证明方法是合理的，但是同时也承认，确认这一结论还需要一段时间，以便人们进行足够的计算进一步证明其正确性。

19.3.1 数独游戏有多少终盘

除了提示数与唯一解的关系，关于数独有多少终盘的问题，也是一个很有趣的话题。很多人尝试用穷举的方法找出所有数独终盘，但是大家显然都低估了这个问题。首先来看一下解空间，在不考虑规则的情况下，81 格单元格填数字有 9^{81} 次方种结果，即使有计算机能进行每秒一千亿次穷举，也需要 62350028689609625069151416960588570612348887285709908335653 年才能穷举出所有解（有兴趣的读者可以用第 17 章介绍的大数类 `CBigInt` 计算一下），也就是说，到那时候才能知道到底有多少个合法的数独终盘。于是，这个问题就变得很无趣了。

但是最近，这个问题又变得有趣了。首先是 2005 年，Bertram Felgenhauer 和 Frazer Jarvis 两个人发表了一篇名为“Enumerating possible Sudoku grids”的文章，文章中介绍了一套方法，通过小范围的穷举加上数独单元格的对称性，计算出一共有 6670903752021072936960 种合法的数独终盘。有兴趣的读者可参考本章参考资料[2]中给出的链接，该网页有证明的方法和相关代码。不过没过多久，有人就发现其实有个网名为“QSCGZ”的家伙在早 2003 年就在谷歌的“rec.puzzles”群中发布了这个结果，答案是一样的，但是没有给出计算方法和推算原理。对此有兴趣的读者可以通过本章参考资料[3]中的链接查到这篇帖子。

www.sudoku.com 网站对这个问题也有很多讨论，大家还是比较认可 6670903752021072936960 这个答案的。实际上，如果排除对称性和数字换位等重复因素，数独终盘一共有 3546146300288 种。知道了数独终盘的个数，那么这么多终盘最终又能产生多少数独题目呢？我们不妨来计算一下，每个数独有 81 个单元格，假设每次挖掉 n 个数字形成一个数独题目，根据排列组合理论，一共有 C_{81}^{n} 种挖法。为了保证数独有唯一解，至少要保留 17 个提示数，也就是说 n 最多只能是 81−17=64。如果每次只挖掉一两个数，肯定会被认为是侮辱大家的智商，因此 n 的最小值也必须是一个合理的数字。本人觉得每行或每列至少要挖掉两个格子的题目才值得动动脑子（尽管这仍然是非常简单的题目），因此设计 n 最小值是 18。这样一来，每个终盘能设计出的数独题目就是 $\sum_{n=18}^{64} C_{81}^{n}$ 个。这只是一个终盘的结果，考虑到终盘的个数，最终能产生的数独游戏是个天文数字。

数独的终盘有如此之多，以至于就算分给全世界 60 亿人解决，每个人都可以分得一万亿个数独终盘。数独爱好者们可能会很泄气，一辈子玩过的数独也只是整个游戏的皮毛而已，但这也正是数独游戏的乐趣所在，不用担心你曾经做过的数独再次耽误你的时间，因为你在有生之年碰上两个相同的数独游戏的概率太小了，你面对的每一个数独都是你以前没有见过的。

19.3.2　史上最难的数独游戏

2013 年 5 月，网络上的各种媒体都在炒作一则新闻：69 岁中国农民三天破解世界最难数独游戏。我曾经关注过这个问题，新闻中提到的世界最难数独游戏据说是芬兰数学家因卡拉（Arto Inkala）花了三个月的时间设计的一个数独局，号称难度系数达到 11，是世界上最难的数独局。评价一个数独游戏的难度没有量化标准，采用不同的评价标准得到的评价结果也不一样。目前有很多评价数独游戏难度的软件，比较有名的是 Nicolas Juillerat 开发的 Sudoku Explainer 和 Bernhard Hobiger 开发的 Hodoku。一般人能够解出的数独游戏难度都在 1 ~ 5 级，Sudoku Explainer 给因卡拉设计的这个数独的难度评价是 10.7。

因卡拉的这个数据题目如图 19-2 所示。

	1	2	3	4	5	6	7	8	9
A	8								
B			3	6					
C		7			9		2		
D		5				7			
E					4	5	7		
F				1				3	
G			1					6	8
H			8	5				1	
I		9					4		

图 19-2　因卡拉设计的“世界最难”数独

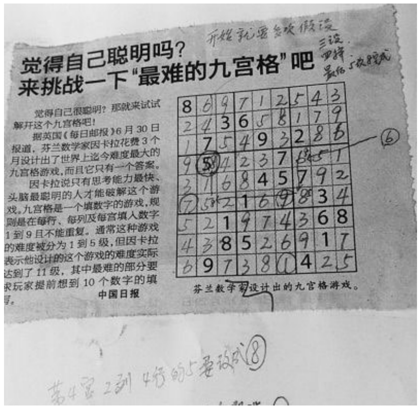


图 19-3　媒体刊登的解题手稿

这个题目确实是有一定的难度，用本文给出的算法进行求解，回溯深度达到了 16。回到刚才的新闻话题，有多事的媒体后来公开了这位老先生解这个题目时的手稿，不过有细心的网友发现，他其实是改动了一个数字才得到了结果。如图 19-3 所示，他将 D2 单元格的 5 改成了 8，于是得到了一个答案。看似微小的改动，却大大降低了这个题目的难度。改动之后，题目由唯一解变成了多解（用本章的算法求解，得到 295 个解）。

后来，包括大学教授在内的众多数独爱好者都给出了正确答案，但是都很难分辨出是自己做的还是计算机做的。

19.4　总结

本章介绍了一种非常流行的数字游戏——数独，并且给出了一种结合候选数列表和相关 20 格深度搜索技术的数独求解算法。应用本算法可以有效地减少穷举过程中的搜索深度和试数次数，是一种比较高效率的算法。当然，本章还讨论了数独终局个数和最难数独游戏这样的八卦话题，希望读者们从数独游戏自动求解算法中体会到更多的乐趣。

19.5 参考资料

[1] Felgenhauer B, Jarvis F. *Enumerating possible Sudoku grids*

[2] http://www.afjarvis.staff.shef.ac.uk/sudoku/bertram.html

[3] https://groups.google.com/forum/#!topic/rec.puzzles/A7pi7S12oFI

第 20 章 华容道游戏

华容道游戏具有开局变化多端、百玩不厌的特点，与七巧板和九连环一起被数学界称为“中国古典智力游戏三绝[①]”。一般人们说华容道游戏的最快解法是 81 步，事实上这样说是不准确的。传统的华容道游戏有很多种开局，比如“横刀立马”“齐头并进”“兵分三路”等，目前已知“横刀立马”开局的最快解法是 81 步。事实上，“齐头并进”开局的最快解法是 60 步，而号称最难开局的“峰回路转”的最快解法是 138 步。

二十世纪六七十年代，中日学者经过努力将“横刀立马”开局的解法提高到 82 步，从 87 步到 82 步用了几十年的时间。但是很快就有美国人提出了 81 步解法，并说这是理论上最快的解法了。正当大家惊呼不解的时候，谜团解开了，原来美国人用计算机搞定了这个问题。外行可能觉得计算机太神奇了，但是我们程序员应该知道，这就是个算法设计的问题。迄今为止，华容道问题还没有找到任何数学理论支持，自然也没有数学解决方法，目前的求解算法基本上都是穷举。再次提醒，不要小看穷举算法，目前还只能靠它了。本章我们就来介绍华容道游戏的解题算法。说实话，自从有了计算机算法解决华容道游戏，人们争相研究最快解法的意义就消失了，但是这个游戏本身还是值得一玩的。

20.1 华容道游戏介绍

华容道游戏的名字据说来源于著名的三国故事“诸葛亮智算华容，关云长义释曹操”。根据小说《三国演义》描述，曹操在赤壁大战中被刘备和孙权的“苦肉计”“火烧连营”打败，被迫由乌林向华容县撤退，在地势险要的华容道与关羽相遇。曹操当时败退得十分狼狈，已经无力与关羽一战，只得哀求关羽放行。关羽念当年曹操的旧日恩情，义释曹操，使其安全回到江陵。华容道游戏的内容正是取自这一典故。华容道游戏虽然有个很中国化的名字，但是它的发明者并不是中国人。美国人 Lewis W. Hardy 在 1909 年就申请了名为“Pennant Puzzle”的美国专利，被称

① 事实上，华容道游戏虽然披着中国文化的外衣，但却是个地地道道的舶来品。这种滑块游戏的首创是美国人，在中国的本土化过程中添加了三国元素。

为华容道游戏的前身。流行于中国的华容道游戏是英国人 John Harold Fleming 在 1932 年发明的（申请了英国专利），在中国流行以后加入了三国背景，成为一款具有中国特色的游戏。

华容道游戏在一个 5×4 的棋盘上布置多名蜀国大将和军士作为棋子，将曹操围困在中间，通过滑动各个棋子，帮助曹操移动到出口位置逃走。图 20-1 就是华容道游戏棋盘的一个展示，使用了经典的“横刀立马”开局。华容道游戏的开局有很多种，用最少的滑动步骤将曹操走出是人们玩这个游戏的乐趣所在。以经典的“横刀立马”开局为例，六十多年前日本人研究出的最快走法是 82 步，后来美国人用计算机穷举得到了最快的走法，只需 81 步。传统的华容道开局布局并不多，但是使用计算机技术以后，更多的有解法的开局被穷举出来，除了传统的“五将四兵一横”布局，还出现了“五将四兵二横”“七将四横三竖（无兵）”以及“七将三横四竖（无兵）”等多种布局方式，解法也趋于多样化。关于华容道游戏的数学原理至今仍然是个迷，20 个格子的棋子游戏竟然有如此多的名堂，也正因为如此，华容道游戏与魔方、独立钻石棋一起被数学界称为三个最不可思议的智力游戏。



图 20-1 典型的华容道游戏棋盘展示

20.2 自动求解的算法原理

目前研究华容道游戏相关的算法，主要关注“有多少种开局”“判断一个局面是否有解”和“求出最优解”三个问题。由于华容道游戏的数学原理仍然没有被研究清楚，因此还没有数学的方法可以解决华容道游戏的求解问题。现在人们用计算机求解华容道问题，基本上都是建立在穷举法基础上的各种算法。用某种精心设计的算法穷举搜索所有可能的解，然后找出滑动步骤最少的解作为最优解，所以“判断一个局面是否有解”实际上和“求出最优解”是联系在一起的。

使用穷举法的关键是要弄清楚穷举的对象是什么。在第 5 章和第 6 章都介绍了穷举法，首先根据问题描述抽象出一个状态，将问题描述演变成对状态的穷举。本章要介绍的算法也不例外，也需要将问题抽象出一个便于计算机处理的可比较、可存储、可转化为结果的东西。因为这是一个棋盘类游戏，我们将其称为“局面”。简单地讲，图 20-1 就是一个局面，它是一个特殊的局面，也就是开局。我们要做的就是将这样的局面转化成数学模型，也就是计算机能够理解的一系列数据结构。如果不能定义局面，穷举就变得无的放矢，如果局面的数学模型设计得不好，会给算法设计带来很大的麻烦。

20.2.1 定义棋盘的局面

棋类游戏的局面一般至少包含两部分内容，分别是棋盘的状态和棋子的状态。对于华容道游

戏而言，可以将 5×4 的格子定义为棋盘，将武将定义为棋子，这样华容道游戏的局面就可以仿照棋类游戏的局面进行定义。先来看看棋盘的定义，棋盘上有 5×4 个格子，每个格子地位相等，没有特殊的格子。每个格子有两个状态，即被某个棋子（武将）或棋子的一部分占用的状态和未被任何棋子占用的空状态。一般可采用二维数组描述棋盘，数组元素的值用 1 和 0 分别表示被占用状态和空状态。为了使每个数组元素能够表示更多的信息，同时简化移动棋子的合法性判断算法，我们对上述棋盘模型中的定义进行扩充。首先是扩充棋盘数组元素值的范围，如果每个数组元素的值是 0，则表示对应的格子是空；如果数组元素的值是非 0 的值；则表示是哪位武将占用了这个位置，其值就是武将对应的编号。其次是给棋盘定义增加一个“边界”。所谓的边界就是将棋盘扩大为 7×6，棋子布局仍然使用中间的 5×4 个格子，四周围绕的格子被赋予一个特殊的值，作为边界类型的格子。中间 5×4 个格子的值要么是 0，要么是某个武将的编号。我们用从 1 开始索引值为每个武将编号，华容道的棋盘上能放置的武将个数最多不会超过 14 个，所以我们用 15（0x0F）作为边界格子的值。增加的这些边界格子其实就是设计计算机算法时常用的“哨兵位”（Guard Bit），哨兵位在算法中的意义就是表示这些位置已经被一个特殊的武将棋子占据了，其他武将棋子不能移动到这里。这是一种常见的算法设计技巧，可以避免数据访问过程中为防止越界而进行的边界保护判断，使得算法能够对移动武将的操作进行一致性处理。以本章介绍的算法为例，如果不采用哨兵位，每次判断一个棋子向各个方向滑动的合法性时，除了判断要移动的方向是否是空的格子之外，还要针对四个方向分别做边界判断。而使用了四周边界的哨兵位后，只需根据滑动方向修正位置，然后判断新位置是否为空位置即可，因为边界格子的值和其他已经被棋子占用的格子的值都是非 0 值。图 20-2 展示了采用这个棋盘设计方案的“横刀立马”开局的棋盘状态。

f	f	f	f	f	f
f	1	2	2	3	f
f	1	2	2	3	f
f	4	5	5	6	f
f	4	8	9	6	f
f	7	0	0	a	f
f	f	f	f	f	f

图 20-2　“横刀立马”开局的棋盘状态

接下来我们看看棋子的定义。虽然扩充定义的棋盘数学模型中暗含了每个武将棋子的信息（棋盘上对应的格子的状态只是武将的编号），可以根据这些信息反向计算出每个武将的位置，但是出于算法效率的原因，我们仍然用一个单独的表来记录每个武将在棋盘上的位置信息，而棋盘数组元素的值就是对应的武将在这个位置表中的索引，这也是算法设计常用的“以空间换时间”的策略。每个武将（棋子）有两个属性，即位置和棋子类型。我们用行和列的值组合成一个二维坐标来表示棋子左上角的位置，棋子其他部分占用的位置则可以根据棋子类型推算出来。棋子类型一共有四种，分别是大方块（2×2）、横长方形（2×1）、竖长方形（1×2）和小方块（1×1），可以用枚举类型来表示这个属性。武将的数据结构定义如下所示：

```
typedef struct tagWARRIOR
{
    WARRIOR_TYPE type;
    int left;
    int top;
}WARRIOR;
```

棋盘采用一个 7×6 的二维数组表示，一个游戏局面的完整定义如下：

```
struct HRD_GAME_STATE
{
    char board[HRD_BOARD_HEIGHT][HRD_BOARD_WIDTH];
    std::vector<WARRIOR> heroes;
    MOVE_ACTION move;
    int step;
    HRD_GAME_STATE *parent;
};
```

这个定义中除了表示棋盘的二维数组之外，还有一个存放武将信息的一维表，其他三个属性是算法设计过程中需要的附加信息，后面会介绍这三个属性的意义。

20.2.2 算法思路

二人对战的棋类游戏一般采用博弈树相关算法处理棋局演化状态的搜索，但是本问题仅仅是棋盘类游戏，棋局的演化和搜索都可以采用更简单的方法。对于一个棋局，如果有一种或多种移动武将的方法，这个棋局就可以演化成一个或多个新的棋局，每个新的棋局又可以根据移动武将的方式演化成更多的棋局。很显然，本问题的棋局搜索空间是一个树状关系空间，对树状空间的搜索既可以采用广度优先搜索，也可以采用深度优先搜索。本书的第 5 章和第 6 章介绍的状态搜索算法都采用了深度优先搜索算法，本章介绍的算法将采用广度优先搜索算法。

我们对算法的要求是：给定一个华容道游戏的开局布局，可以得到这个开局的所有解决方法（包括最少武将移动步骤的方法）以及相应的武将移动步骤，要求算法具有通用性，能处理任何一种开局的华容道游戏。为此我们在棋局状态定义中额外增加了三个属性，分别是 move、step 和 parent。move 是当前棋局对应的动作，step 记录移动的步骤，也就是记录 move 动作是第几次移动武将，parent 是当前棋局的“父棋局”，可以这样理解 parent 对应的棋局采用 move 动作后得到当前棋局。添加 parent 的目的是在搜索到一个能让曹操成功逃脱的棋局时，通过 parent 能回溯整个武将移动过程，输出移动武将的步骤。

能推动棋局变化的事件是在棋盘上移动武将的动作（Action），对于一个棋局来说，一个动作应该包含两个信息，其一是动作是哪个武将产生的，也就是说移动的是哪个武将；其二是动作的方向，根据华容道游戏的规则，合法的移动方向只有上、下、左、右四个方向。动作的定义可以这样实现：

```
typedef struct tagMOVE_ACTION
{
    int heroIdx;
    int dirIdx;
}MOVE_ACTION;
```

我们定义了棋局，定义了武将移动的动作，剩下工作的工作就是找出移动武将，驱动棋局状态变化的算法。移动武将，实际上就是调整武将的 left 和 top 位置，为了使调整算法能够做到不依赖具体的方向，一致性处理向四个方向的移动，我们再次使用了方向数组技巧。首先将方向的定义分解为横向和纵向的移动量，如下所示：

```
typedef struct tagDIRECTION
{
    int hd;
    int vd;
}DIRECTION;
```

所谓的向上移动棋子，其实就是在保持棋子列方向位置不变，将棋子整体行方向减 1。同理，向其他方向移动可以分别描述为对相应方向行或列坐标的加 1 或减 1 计算。最后这样定义向四个方向移动的方向数组：

```
DIRECTION directions[MAX_MOVE_DIRECTION] = { {0, -1}, {1, 0}, {0, 1}, {-1, 0} };
```

这样做的好处是循环一遍这个方向数组，将每个方向的横向和纵向移动量与武将的当前位置叠加，即可实现向四个方向移动武将的功能，避免了 if(向上)…if(向下)…这样的代码。

搜索算法的核心是通过动作推动棋局的转化，对于一个棋局的当前状态，遍历所有的武将棋子，判断它们能否移动，如果能够移动，则尝试移动它，使得棋局发生变化得到一个新的棋局。广度优先搜索算法通常用一个线性表存储所有棋局，按照顺序对表中每一个棋局进行搜索，如果找到新棋局则添加到棋局表的尾部，重复这个搜索过程直到结束，结束条件是搜索到期望的结果棋局或搜索完最后一个棋局也没有再产生新的棋局（棋局表已经空了）。

20.3 自动求解的算法实现

20.2 节给出了搜索算法的原理，算法原理虽然简单，但是实现起来还有许多具体的问题需要明确。首先是重复棋局的问题。搜索过程中肯定会出现之前已经处理过的棋局，重复处理会降低算法的效率，对这样的棋局应该丢弃。要做到这一点，就需要设计一套存储和比较棋局的方法。

其次是棋盘状态的左右镜像的问题。所谓的左右镜像问题，就是两个棋局虽然武将的位置不一样，但是如果忽略武将的名字信息，单纯从形状上看是左右对称的镜像结构，图 20-3 是左右镜像的一种常见情况，图 20-4 则是另一种左右镜像的常见情况。对于华容道游戏来说，这种左右镜像的情况对于滑动棋子寻求结果的影响是一样的，也就是说，如果一个棋局存在一个 80 步的解，则它的左右镜像棋局也存在一个 80 步的解，而且相同形状的棋子的移动步骤和顺序完全一样（当然，对武将来说是相同形状的不同武将）。一般华容道游戏的求解算法都要处理左右镜像的情况，将左右镜像视为相同的棋局而丢弃掉。要做到这一点，就需要对棋局进行模式定义并识别出左右镜像的情况。

最后是武将的连续滑动问题。根据华容道游戏的规则，武将的连续滑动（1×1 的小方块棋子比较容易出现）被视作走一步。针对这个规则，对武将的每次位置移动都需要做特殊的判断，滑动一次和滑动两次虽然对最后的结果输出都是一步，但是中间会得到两个不同的棋局，对这两个棋局都要进行处理。

本节介绍算法实现时将会遇到这些问题，当然还包括其他问题，接下来就分别介绍如何处理

这些问题，并给出解决这些问题的算法。

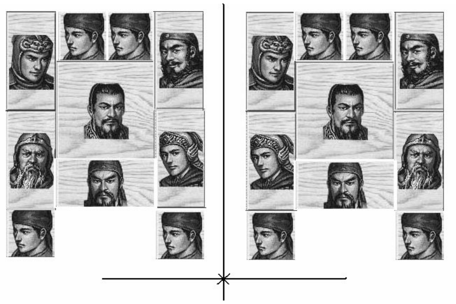


图 20-3 左右镜像的常见情况之一

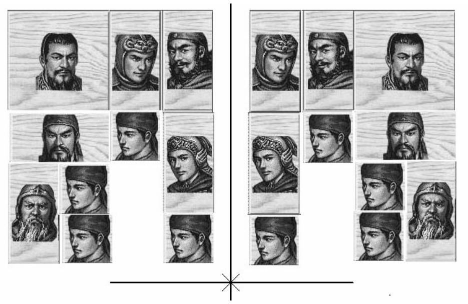


图 20-4 左右镜像的常见情况之二

20.3.1 棋局状态与 Zobrist 哈希算法

华容道游戏中棋子的位置和棋盘的状态是标识一个棋局有别于另一个棋局的关键数据。要进行棋局的比较判断是否是重复处理过的棋局，就必须能够对这两个关键数据进行存储、比较和转换。既然我们已经在 20.2.1 节设计了棋局的数据结构，那就应该可以通过逐个比较这些数据结构中的属性的值来判断两个棋局是否相同。这样做确实可以，但并不是高效的算法。在一些复杂的棋类游戏中，棋局的产生数以亿计，直接对缤纷复杂的数据逐个比较是不能接受的。

1. Zobrist 哈希算法原理

棋类游戏中通常采用各种哈希算法对棋局进行处理，得到一个可在 $O(1)$时间复杂度内判断是否是重复棋局的哈希表。Zobrist 哈希算法是一种适用于棋类游戏的棋局编码方式，以其发明者 Albert L. Zobrist 的名字命名。Zobrist 哈希算法通过建立一个特殊的置换表，对棋盘上每一个位置的所有可能状态赋予一个绝不重复的随机编码，通过对不同位置上的随机编码进行异或计算，实现在极低冲突率的前提下将复杂的棋局编码为一个整数类型哈希值的功能。

Zobrist 哈希算法的哈希编码步骤如下。

(1) 识别出棋局的最小单位（格子或交叉点），确定每个最小单位上的所有可能的状态数。以华容道的棋局为例，最小单位就是 20 个小格子，每个格子有 5 种状态，分别是空状态、被横长方形占据、被竖长方形占据、被小方格占据和被大方格占据。

(2) 为每个单位上的所有状态都分配一个随机的编码值。棋类游戏一般需要“行数×列数×状态数”个状态，以华容道游戏为例，编码值采用 32 位，需要为 5×4×5=100 个状态分配编码值。

(3) 对指定的棋局，对每个单位上的状态用对应的编码值（随机数）做异或运算，最后得到一个哈希值。

以上第(1)步和第(2)步是准备阶段，可以实现计算并分配好，只有第 3 步是需要对每个棋局进行编码计算。用 Zobrist 算法产生的编码值是个随机数，表面上看起来好像和棋局没有什么关

系，但是如果棋子被移动过，相关的一个或多个最小单位上的状态就会变化，于是对应的编码值也就变化了，最终会反映到棋局的哈希值发生变化。也就是说，只要一个棋子发生变动，最终得到的棋局哈希值也会变化。

Zobrist 哈希算法有两大优点，第一个优点是冲突概率小，只要随机编码值的范围够大，棋局哈希冲突的概率非常小，实际应用中基本上不考虑冲突的情况（最多就是出个昏招输掉一局棋）。第二个优点是棋局发生变化时，不必对整个棋局重新计算哈希值，只需要计算发生变化的那些最小单元的状态变化即可，对棋类游戏算法的搜索效率来说，这是一个非常诱人的红利。举个例子，如果把华容道游戏的一个"小卒"从 A 位置移到 B 位置，棋局发生了变化，但是不需要重新计算整个棋盘上的状态，只需要计算 A 和 B 两个位置的变化。首先，A 位置的状态从 1（1×1 的小方块的类型值）变化成空状态（0），这时候只需要将 A 位置上状态 1 对应的编码值与棋局的哈希值再做一次异或运算。根据异或运算的特点，就相当于"小卒"从 A 位置上"删除了"，然后再将 A 位置上空状态对应的编码值与棋局的哈希值做异或运算，相当于将 A 位置改变为空状态。对 A 位置处理完，继续对 B 位置进行处理，B 位置的处理与 A 位置的处理一样，先将 B 位置上空状态对应的编码值与棋局的哈希值做一次异或运算，再将 B 位置上状态 1 对应的编码值与棋局的哈希值做一次异或运算。原来棋局的哈希值经过四次异或运算后得到的值就是新棋局的哈希值，这就是 Zobrist 哈希算法增量计算的优点。实际上，我们会将棋盘的空状态的值设为 0，这样一来，与空状态进行的两次异或状态就没有必要了（与 0 异或不改变原值），最终只需两次异或运算就可以得到新棋局的哈希值。

2. Zobrist 哈希算法实现

实现 Zobrist 哈希算法首先要定义编码表，编码表是针对棋局定义的。根据上一节的描述，很显然这是一个三维的表，为了更清晰地表达这个表的结构，我们将这个三维表分成最小单元定义和最小单元的状态定义两个数据结构，如下所示：

```
typedef struct tagCellState
{
    int value[MAX_WARRIOR_TYPE];
}CELL_STATE;

typedef struct tagZobristHash
{
    CELL_STATE key[HRD_GAME_ROW][HRD_GAME_COL];
}ZOBRIST_HASH;
```

对于复杂的棋类游戏，一般认为采用 64 位整数编码是安全的，但是对于华容道这样简单的局面，我们觉得用 32 位整数表示随机编码就足够了，因此 value 的定义用的是无符号整数。

计算整个棋局的哈希值的过程就是首先初始化哈希值为 0，然后对 20 个棋盘格子逐个处理，根据棋盘格子的武将编号信息获取武将的类型（也就是棋盘格子的状态），根据武将类型获取该类型对应的编码值，用此编码值参与哈希值进行异或运算，处理完所有期盼格子后的哈希值就是最终的结果。完整的算法实现如 GetZobristHash()函数所示。

```
unsigned int GetZobristHash(ZOBRIST_HASH *zob_hash, HRD_GAME_STATE *state)
{
    unsigned int hash = 0;
    const std::vector<WARRIOR>& heroes = state->heroes;
    for(int i = 1; i <= HRD_GAME_ROW; i++)
    {
        for(int j = 1; j <= HRD_GAME_COL; j++)
        {
            int index = state->board[i][j] - 1;
            int type = (index != 0) ? heroes[index].type : 0;
            hash ^= zob_hash->key[i - 1][j - 1].value[type];
        }
    }
    return hash;
}
```

Zobrist 哈希算法的一个主要优点就是当棋局发生变化时，只需要计算变化的部分就可以得到新棋局的哈希值。这一点在 GetZobristHashUpdate()函数中得到体现，将编号为 heroIdx 的武将棋子向 dirIdx 指定的方向移动一步后，这个函数只重新计算移动武将棋子所影响的几个位置就可以得到新的棋局的哈希值。

```
unsigned int GetZobristHashUpdate(ZOBRIST_HASH *zob_hash, HRD_GAME_STATE *state, int heroIdx, int
dirIdx)
{
    unsigned int hash = state->hash;
    const WARRIOR& hero = gameState->heroes[heroIdx];
    const DIRECTION& dir = directions[dirIdx];

    switch(hero.type)
    {
        ...
    case HT_VBAR:
        //原始位置的处理：
        hash ^= zob_hash->key[hero.left][hero.top].value[hero.type];
        hash ^= zob_hash->key[hero.left][hero.top + 1].value[hero.type];
        hash ^= zob_hash->key[hero.left][hero.top].value[0]; //0是空状态
        hash ^= zob_hash->key[hero.left][hero.top + 1].value[0];
        //新位置的处理：
        hash ^= zob_hash->key[hero.left + dir.hd][hero.top + dir.vd].value[0];//0是空状态
        hash ^= zob_hash->key[hero.left + dir.hd][hero.top + dir.vd + 1].value[0];
        hash ^= zob_hash->key[hero.left + dir.hd][hero.top + dir.vd].value[hero.type];
        hash ^= zob_hash->key[hero.left + dir.hd][hero.top + dir.vd + 1].value[hero.type];
        break;
        ...
    }

    return hash;
}
```

20.3.2 重复棋局和左右镜像的处理

重复棋局的判断依赖于 Zobrist 哈希算法对棋局计算出的哈希值。搜索算法用一个集合存放

已经处理过的棋局的哈希值，集合中的元素没有重复，我们可以利用 STL 的 std::set 容器。搜索算法在搜索棋局的过程中，每产生一个新的棋局，就调用 AddNewStatePattern()函数判断是否是重复的棋局。这个函数就是查这个哈希值的集合，如果已经存在相同的棋局，则返回 false，否则就返回 true，并将新棋局的哈希值加入到这个集合中，同时新棋局加入到棋局队列中，这个棋局队列是广度优先搜索算法需要的搜索队列。

```
bool AddNewStatePattern(HRD_GAME& game, HRD_GAME_STATE* gameState)
{
    unsigned int l2rHash = GetZobristHash(&zob_hash, gameState);
    if(game.zhash.find(l2rHash) == game.zhash.end())
    {
        game.zhash.insert(l2rHash);
#if NO_LR_MIRROR_ALLOW
        unsigned int r2lHash = GetMirrorZobristHash(&zob_hash, gameState);
        game.zhash.insert(r2lHash);
#endif
        game.states.push_back(gameState);

        return true;
    }

    return false;
}
```

AddNewStatePattern()函数内部对左右镜像的情况也做了处理，调用 GetMirrorZobristHash()函数计算镜像棋局的哈希值。求镜像棋局的哈希值不需要先求出镜像棋局再计算哈希值，可以用 Zobrist 哈希算法的特性，通过对调左右两列的状态编码值的方式，直接在原棋局上计算镜像棋局的哈希值，请看 GetMirrorZobristHash()函数的实现代码。

```
unsigned int GetMirrorZobristHash(ZOBRIST_HASH *zob_hash, HRD_GAME_STATE *state)
{
    unsigned int hash = 0;
    const std::vector<WARRIOR>& heroes = state->heroes;
    for(int i = 1; i <= HRD_GAME_ROW; i++)
    {
        for(int j = 1; j <= HRD_GAME_COL; j++)
        {
            int index = state->board[i][j] - 1;
            int type = (index >= 0 && index < heroes.size()) ? heroes[index].type : 0;

            //(HRD_GAME_COL - 1) - (j - 1))
            hash ^= zob_hash->key[i - 1][HRD_GAME_COL - j].value[type];
        }
    }
    return hash;
}
```

GetMirrorZobristHash()函数与 GetZobristHash()函数的区别就是将 key 的列下标定位由 j - 1 改成了 HRD_GAME_COL - j，相当于将第 1 列的状态编码值和第 4 列的状态编码值交换，将第 2 列的状态编码值与第 3 列的状态编码值交换，这样交换后直接对当前棋局进行计算，得到的就是其

镜像棋局的哈希值。NO_LR_MIRROR_ALLOW 是编译控制，决定算法搜索过程中是否排除镜像棋局，如果要排除镜像棋局,就需要将 GetMirrorZobristHash()函数计算出来的镜像棋局的哈希值也加入到已经处理过的棋局哈希值集合中，从而避免在后序的搜索过程中再重复处理到镜像棋局。

20.3.3 正确结果的判断条件

很显然，当代表曹操的大方块棋子出现在棋盘下部的中央缺口位置的时候，游戏就结束了。根据之前对武将的定义，这个条件其实就是代表曹操的棋子的左上角位置位于[1, 3]时。现在问题是，哪个棋子代表曹操？可以约定武将数据初始化时第一个武将总是曹操，也可以使用一个属性保存曹操的棋子编号，以便可以随时找到曹操（传说中的“说曹操，曹操到”）。我们的算法采用第二种方式，在 HRD_GAME 的定义中增加一个表示曹操编号的属性。2.3.2 节介绍 AddNewStatePattern()函数的时候已经引用了这个定义，HRD_GAME 的完整定义如下：

```
typedef struct tagHRD_GAME
{
    std::string gameName;
    std::vector<std::string> heroNames;
    char caoIdx;
    std::deque<HRD_GAME_STATE *> states;
    std::set<unsigned int> zhash;
    int result;
}HRD_GAME;
```

gameName 和 heroNames 的定义是为了使得输出结果更具趣味性，caoIdx 就是曹操棋子的编号，states 是广度优先搜索算法需要的棋局队列，zhash 是棋局哈希表，result 记录搜索算法结束后找到了几种正确的解。

完成以上数据结构定义，就可以用 IsEscaped()函数进行是否得到正确游戏结果的判断了，这个函数的实现非常简单，就是判断曹操棋子的左上角坐标是否是[1, 3]。


```
bool IsEscaped(HRD_GAME& game, const HRD_GAME_STATE* gameState)
{
    return (gameState->heroes[game.caoIdx - 1].left == CAO_ESCAPE_LEFT)
            && (gameState->heroes[game.caoIdx - 1].top == CAO_ESCAPE_TOP);
}
```

20.3.4 武将棋子的移动

这一节我们讨论华容道游戏棋盘上武将棋子的移动问题。移动武将棋子相当于产生了新状态，所以移动棋子也是棋局搜索算法的基础。从数据的角度理解棋子的移动，就是将棋盘中武将所在位置的信息清除，然后在新位置上设置武将的信息。这就是武将棋子移动算法的实现方法，在棋局上移动棋子需要两个信息，一个是武将的编号，另一个是移动方向。

移动武将棋子之前首先要判断能否移动这个武将棋子。判断的依据有两个，首先，不能移出边界；其次，新位置上不能有其他武将棋子。这很容易理解，但是编写算法实现要考虑周到。之

前我们在设计棋盘时在棋盘四周设置了哨兵位，这样我们在设计算法时就不需要再单独判断每次移动是否超出边界了，只需关注新位置是否被其他武将棋子占用即可。新位置如果是 0，表示空位置，非 0 则表示被其他武将棋子（包括边界）占用。但是一点需要注意的是，在棋盘上移动棋子，多数情况下都只能移动一个格子的位置，对块头比较大、需要占用多个格子的武将棋子，需要考虑位置重叠的情况。以横长方形的棋子为例，如果向右移动一个格子，横长方形右边格子和左边格子的判断条件有点区别，右边格子需要判断新位置是否是空，左边格子因位置重叠不需要做判断。如果向左移动则刚好相反，左边格子需要判断新位置是否是空，又边格子因位置重叠不需要做判断。那么是否需要对不同的移动方向做不同的处理呢？答案是否定的，因为如果这样做就违背了我们当初将移动方向设计为数组的初衷。因为每个武将的编号不同，这个给算法的重叠判断提供了依据，对于横长方形的两个格子不需要根据方向做区别处理，只要判断如果新位置是 0，表示新位置是空可以移动，如果新位置是当前武将的编号，则说明是位置重叠，也可以移动。具体代码实现请看 CanHeroMove()函数。

```
bool CanHeroMove(HRD_GAME_STATE* gameState, int heroIdx, int dirIdx)
{
    int cv1,cv2,cv3,cv4;
    bool canMove = false;
    const WARRIOR& hero = gameState->heroes[heroIdx];
    const DIRECTION& dir = directions[dirIdx];

    switch(hero.type)
    {
    ...
    case HT_VBAR:
        cv1 = gameState->board[hero.top + dir.vd + 1][hero.left + dir.hd + 1];
        cv2 = gameState->board[hero.top + dir.vd + 2][hero.left + dir.hd + 1];
        canMove = (cv1 == BOARD_CELL_EMPTY || cv1 == heroIdx) && (cv2 == BOARD_CELL_EMPTY || cv2 ==
          heroIdx);
        break;
    ...
    }

    return canMove;
}
```

MoveHeroToNewState()函数移动武将棋子产生新棋局，这个函数的核心操作就是判断是否能移动棋子，如果能移动棋子就产生一个新状态，清除原位置上的棋子信息，在新位置上设置棋子信息，其他的代码都是处理辅助数据的。

```
HRD_GAME_STATE* MoveHeroToNewState(HRD_GAME_STATE* gameState, int heroIdx, int dirIdx)
{
    if(CanHeroMove(gameState, heroIdx, dirIdx))
    {
        HRD_GAME_STATE* newState = new HRD_GAME_STATE;
        if(newState != NULL)
        {
            CopyGameState(gameState, newState);
            WARRIOR& hero = newState->heroes[heroIdx];
```

```
            const DIRECTION& dir = directions[dirIdx];

            ClearPosition(newState, hero.type, hero.left, hero.top);
            TakePosition(newState, heroIdx, hero.type, hero.left + dir.hd, hero.top + dir.vd);
            hero.left = hero.left + dir.hd;
            hero.top = hero.top + dir.vd;

            newState->step = gameState->step + 1;
            newState->parent = gameState;
            newState->move.heroIdx = heroIdx;
            newState->move.dirIdx = dirIdx;
            return newState;
        }
    }

    return NULL;
}
```

根据华容道游戏规则，对一个武将棋子连续移动只算一步。如果搜索算法只是确定指定的局面是否有解，这个规则实际上对算法没有影响，如果需要输出移动的步骤并计算最小移动步骤，则必须考虑这个问题。在每一步移动成功以后，继续对该棋子尝试移动，但是移动的方向有限制，不能向原方向移动。

```
void TryHeroContinueMove(HRD_GAME& game, HRD_GAME_STATE* gameState, int heroIdx, int lastDirIdx)
{
    int d = 0;
    /*向四个方向试探移动*/
    for(d = 0; d < MAX_MOVE_DIRECTION; d++)
    {
        if(!IsReverseDirection(d, lastDirIdx)) /*不向原方向移动*/
        {
            HRD_GAME_STATE *newState = MoveHeroToNewState(gameState, heroIdx, d);
            if(newState != NULL)
            {
                if(AddNewStatePattern(game, newState))
                {
                    newState->step--;
                }
                else
                    delete newState;

                return;
            }
        }
    }
}
```

判断两个方向是否是反方向，不需要写一堆 if 语句，再次体现了方向数组的好处：

```
bool IsReverseDirection(int dirIdx1, int dirIdx2)
{
    return ( ((dirIdx1 + 2) % MAX_MOVE_DIRECTION) == dirIdx2);
}
```

20

20.3.5 棋局的搜索算法

华容道游戏的求解过程就是棋局的搜索过程，搜索的过程其实也是棋局生成的过程。移动一个棋子会使棋盘状态产生一个新棋局，而移动另一个棋子则会产生另一个不同的新棋局。此外，对于同一个棋子，向不同的方向移动也会产生不同的新棋局。搜索算法开始的时候，棋局队列中只有一个元素，就是游戏的开局状态。搜索算法每次从棋局队列中取出一个棋局，首先判断是否是结束状态，如果是结束状态，就输出结果，否则就对这个棋局尝试各种移动武将棋子的操作，新产生的棋局如果之前没有出现重复的新棋局，就加入到棋局队列。

```
bool ResolveGame(HRD_GAME& game)
{
    int index = 0;
    while(index < static_cast<int>(game.states.size()))
    {
        HRD_GAME_STATE *gameState = game.states[index];
        if(IsEscaped(game, gameState))
        {
            game.result++;
            OutputMoveRecords(game, gameState);
        }
        else
        {
            SearchNewGameStates(game, gameState);
        }

        index++;
    }

    return (game.result > 0);
}
```

SearchNewGameStates()函数对棋盘上的武将和可能的移动方向进行组合枚举，驱动搜索算法产生新的棋局状态，这是一个两重循环，也是组合类枚举惯用的方法：

```
void SearchNewGameStates(HRD_GAME& game, HRD_GAME_STATE* gameState)
{
    for(int i = 0; i < static_cast<int>(gameState->heroes.size()); i++)
    {
        for(int j = 0; j < MAX_MOVE_DIRECTION; j++)
        {
            TrySearchHeroNewState(game, gameState, i, j);
        }
    }
}
```

TrySearchHeroNewState()函数尝试对编号为 i 的武将向 j 指定的方向移动，如果可以通过移动棋子产生新的棋局（MoveHeroToNewState()函数负责做这个工作），就根据游戏规则继续尝试能否连续移动。如果新棋局是重复棋局（AddNewStatePattern()函数负责做这个判断）或移动无法在这个方向移动，就忽略这个武将棋子和移动方向的组合。

```
void TrySearchHeroNewState(HRD_GAME& game, HRD_GAME_STATE* gameState, int heroIdx, int dirIdx)
{
    HRD_GAME_STATE *newState = MoveHeroToNewState(gameState, heroIdx, dirIdx);
    if(newState != NULL)
    {
        if(AddNewStatePattern(game, newState))
        {
             /*尝试连续移动，根据华容道游戏规则，连续的移动也只算一步*/
            TryHeroContinueMove(game, newState, heroIdx, dirIdx);
            return;
        }

        delete newState;
    }
}
```

至此，搜索算法都完整了，结合之前的游戏和棋局定义，就可以求解各种华容道游戏开局了。

20.4 总结

最后不出所料，我们的算法找到了“横刀立马”开局的最快 81 步解法，当然，还有“指挥若定”的 73 步解法，“比翼横空”的 28 步解法，等等。广度优先搜索算法有助于快速找到解决步骤最少的解法，通常第一个输出的结果就是最快的解决方法。

为了使得算法对华容道游戏的结果输出更有趣味，我们的算法采用了单独的数组 `heroes` 存放武将信息，事实证明这极大地影响了算法的速度。新棋局产生时复制棋局数据操作是个瓶颈，可以对此做出优化，比如将武将信息压缩到一个 32 位整数中，直接存放在 `board` 中（需要将 `board` 改成 `int` 类型），有兴趣的读者可自行优化这个算法。

20.5 参考资料

[1] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[2] 维基百科：http://zh.wikipedia.org/wiki/华容道

[3] Surhone L M, Timpledon M T, Marseken S F. *Zobrist Hashing*. Vdm Publishing House, 2010

第 21 章 A*寻径算法

我最初接触计算机是从游戏开始的，最早是 DOS 时代的 RPG 游戏（Role-Playing Game，角色扮演游戏）。只需鼠标一点，游戏中的人物或精灵就会绕过各种障碍物，沿着一条最近的道路到达指定的位置。那时候我对各种算法没有概念，一直很好奇这是怎么实现的。我也模仿着做了一个可以用鼠标控制精灵在地图上移动的小程序，但是可耻地使用了类似迷宫游戏的穷举算法，后来才知道原来大家都用 A*算法。A*算法其实是一类启发式搜索算法的基础，传统上用作寻径（寻路）算法，但是 A*算法的思想并不仅限于游戏中的寻径算法。

A*算法虽然名字很神秘，但是算法的原理和实现都很简单。本章我们将介绍 A*算法，作为对比，我们首先会介绍 Dijkstra（迪杰斯特拉）算法。游戏中的寻径算法一般不会使用 Dijkstra 算法，但是作为一种不带任何启发思想的广度优先搜索算法，刚好可以和 A*算法做个对比。我们编写了一个寻径算法对比演示程序，在一个 16×16 个小方格组成的模拟地图上用图示的方法直观地展示各种算法的搜索方式和搜索效率，以及各种距离评估函数对 A*算法的影响。

21.1 寻径算法演示程序

Dijkstra 算法与 A*算法有很大的差异，A*算法比较适合用于二维平面地图上的寻径算法，如果用小方格模拟地图，A*算法的结果可以直接输出成小方格的状态，而 Dijkstra 算法则需要做一些转换，需要将小方格描述的二维平面地图转化成带权有向图。设计这个演示程序并不是本章的重点，但是为了输出的效果，我们需要对算法定义的数据结构进行调整，使我们给出的算法实现能够输出符合演示程序要求的数据结构和数据。

在一个 16×16 个小方格组成的模拟地图上，每个小方格有不同的状态、类型和标志，我们用不同的颜色表示这些属性。我们给出的小方格属性定义 CELL 如下：

```
typedef struct tagCell
{
    int node_idx;
    int type; //0:normal,1:mark,2:wall,3:source,4:target
    bool inPath;
```

```
    bool processed;
}CELL;
```

type 是 0 表示这是一个普通的小方格，1 表示需要特殊标记，2 表示这个小方格代表的是类似墙一样的障碍物，3 表示这个小方格是寻径的起点，4 表示这个小方格是寻径的终点。inPath 表示这个小方格是否在最后找到的最短路径上，processed 表示这个小方格是否被搜索算法处理过。每种寻径算法最后的结果都输出在 GRID_CELL 中，以便演示程序能够以一致的方式显示各种算法的结果。

```
typedef struct tagGridCell
{
    CELL cell[N_SCALE][N_SCALE];
}GRID_CELL;
```

对于 Dijkstra 算法，还需要将以矩阵方式描述的地图（小方格）状态转化为带权有向图。转化的原则就是小方格矩阵中每个小方格视作带权有向图中的一个顶点（被标记为障碍物的小方格不作为顶点），两个直接相邻的小方格（顶点）视作有一条权为 1 的边将它们相连。如果用邻接矩阵的方式描述带权有向图，则 Dijkstra 算法所用到的数据结构可描述为：

```
typedef struct tagDijkstraGraph
{
    std::vector<GNODE> nodes;
    int adj[N_NODE][N_NODE];
    int prev[N_NODE];
    int dist[N_NODE];
    int source;
    int target;
}DIJKSTRA_GRAPH;
```

其中 nodes 是顶点集合，adj 是邻接矩阵，其他 4 个属性是与 Dijkstra 算法相关的变量，21.2 节具体介绍 Dijkstra 算法时再对它们做详细说明。

对于 A*算法，本身就适合用矩阵方式描述地图，不需要转化为有向图，因此用于 A*算法的数据结构可直接描述为：

```
typedef struct tagAStarGraph
{
    int grid[N_SCALE][N_SCALE];
    std::multiset<ANODE, compare> open;
    std::vector<ANODE> close;
    ANODE source;
    ANODE target;
}ASTAR_GRAPH;
```

其中 grid 直接描述这个小方格矩阵，其他 4 个属性是与 A*算法有关的变量，21.3 节具体介绍 A*算法时再对它们做详细说明。

21.2 Dijkstra 算法

Dijkstra 算法是典型的单源最短路径算法，任何一本介绍图论或数据结构的书都会介绍这种

算法。Dijkstra 算法适用于求解没有负权边的带权有向图的单源最短路径问题，所谓的单源可以理解为一个出发点，Dijkstra算法可以求得从这个出发点到图中其他顶点的最短路径。由于Dijkstra算法使用广度优先搜索策略，它可以一次得到所有点的最短路径。Dijkstra 算法在各种地图软件中得到了广泛的应用，假如我们把一个地区的交通网用带权有向图表示，其中城市就是图的顶点，边就是连接城市之间的道路，边的权就是城市之间的距离，就可以用 Dijkstra 算法求解从一个城市到另一个城市的最短路径。由此可见，地图软件中的这个实用功能背后就是简单的 Dijkstra 算法在支撑，本节我们来介绍一下 Dijkstra 算法。

21.2.1　Dijkstra 算法原理

设 G=(V,E)是一个带权有向图，其中 s 是起始点。Dijkstra 算法的思想就是用一个表存放当前找到的从 s 到每个顶点 v_i 的最短路径，称其为 dist 表。dist 表的初始状态是 dist[s]=0，若存在与 s 直接相连的顶点 m，则记录 dist[m]=$W(s, m)$，其中 $W(s, m)$就是连接 s 和 m 的边的权。对于其他与 s 不直接相连的顶点 v_i，记录 dist[v_i]=+∞。Dijkstra 算法的基本操作就是用广度优先搜索策略处理每一个顶点，对与之相关的边进行拓展。扩展边的方法是：如果存在一条从 u 到 v_i 的边，那么从 s 到 v_i 的最短路径可以通过将边 $W(u, v_i)$添加到尾部来拓展一条从 s 到 v_i 的路径。这条路径的长度是 dist[u] + $W(u, v_i)$，如果这个值比目前已知的 dist[v_i] 的值要小，则使用这个新值来替代当前 dist[v_i] 的值，如果这个值比目前已知的 dist[v_i] 的值大，则不做任何动作。

21.2.2　Dijkstra 算法实现

Dijkstra 算法在搜索过程中需要维护两个顶点的集合 S 和 Q，集合 S 中存放所有已知 dist[v_i]都已经是最短路径的顶点，其余的顶点都在集合 Q 中。初始时 S 集合为空，算法每次从集合 Q 中选择一个顶点 u，其距离 dist[u]的值最小（起点 s 总是被第一个选中，因为 dist[s]的初始状态总是 0），将 u 从 Q 中移到 S 中，然后对每一条与 u 相连的边 $W(u, v_i)$进行扩展，具体的算法步骤如下。

(1) 初始化集合 S 和 Q，设起点 dist[s]=0，并将其他顶点的 dist[v_i]设为无穷大；

(2) 从 Q 中选择一个 dist[u]值最小的顶点 u，将 u 从集合 Q 中移到集合 S 中；

(3) 以 u 为当前顶点，修改 Q 中与 u 相连的顶点的距离。修改的方法是：对于集合 Q 中每一个与 u 相连的顶点 v_i，如果从起点 s 经 u 到的 v_i 距离 dist[u] + $W(u, v_i)$的值小于当前 v_i 的距离 dist[v_i]，则将 dist[v_i]的值修正为 dist[u] + $W(u, v_i)$的值，同时将顶点 v_i 的前驱顶点记为 u；

(4) 重复步骤(2)和(3)，直到集合 Q 为空。

第(3)步记录顶点的前驱顶点操作不是算法的必需内容，记录前驱顶点的目的仅仅是为了能够回溯每条最短路径的顶点连接关系。根据以上分析，Dijkstra 算法的实现如下：

```
void Dijkstra(DIJKSTRA_GRAPH *graph)
{
    std::set<int> S,Q;

    for(int i = 0; i < graph->nodes.size(); i++)
```

```
    {
        graph->dist[i] = graph->adj[graph->source][i];
        graph->prev[i] = (graph->dist[i] == MAX_DISTANCE) ? -1 : graph->source;
        Q.insert(i);
    }
    graph->dist[graph->source] = 0;

    while(!Q.empty())
    {
        int u = Extract_Min(graph, Q);
        S.insert(u);

        for(auto it = Q.begin(); it != Q.end(); ++it)
        {
            int v = *it;
            if((graph->adj[u][v] < MAX_DISTANCE) //小于 MAX_DISTANCE 表示有边相连
                && (graph->dist[u] + graph->adj[u][v] < graph->dist[v]))
            {
                graph->dist[v] = graph->dist[u] + graph->adj[u][v]; //更新 dist
                graph->prev[v] = u;   //记录前驱顶点
            }
        }
    }
}
```

DIJKSTRA_GRAPH 数据结构的定义 21.1 节已经给出，dist 和 prev 两个属性的作用就是记录当前顶点的最短距离和当前节点在最短路径上的前驱节点。Dist[v]表示编号为 *v* 的顶点与源点的最小距离，prev[v]存放的是 *v* 在这条最短路径上的前驱节点，逐次遍历 prev[v]直到达到源点，就能依次得到这条最短路径上的每个顶点，21.2.3 节的 UpdateCellInfo()函数就展示了通过 prev[v]逐次得到最短路径的方法。Extract_Min()函数从集合 *Q* 中找到 dist 值最小的一个顶点，从集合 *Q* 中删除这个顶点并返回这个顶点。Extract_Min()函数的实现非常简单，大家可以从本章的随书代码中找到它的代码。

21.2.3 Dijkstra 算法演示程序

21.2.2 节给出的算法结束后，集合 *S* 中是所有搜索过的顶点，通过 UpdateCellInfo()函数将其转换成 GRID_CELL 数据结构，就可以将 Dijkstra 算法的结果直观地展示出来。图 21-1 是在地图上没有障碍物的情况下 Dijkstra 算法的寻径结果，可以明显地看到广度优先搜索的过程，从源点开始向外层层搜索，直到“碰到”终点为止。根据算法原理，有灰色圆点标识的小方块就是从源点到目标点的最短路径。

```
void UpdateCellInfo(DIJKSTRA_GRAPH *graph, std::set<int>& S, GRID_CELL *gc)
{
    for(auto it = S.begin(); it != S.end(); ++it)
    {
        GNODE node = graph->nodes[*it];
        gc->cell[node.i][node.j].processed = true;
    }
```

```
        int u = graph->target;
        while(u != -1)
        {
            GNODE node = graph->nodes[u];
            gc->cell[node.i][node.j].inPath = true;
            u = graph->prev[u];
        }
    }
```

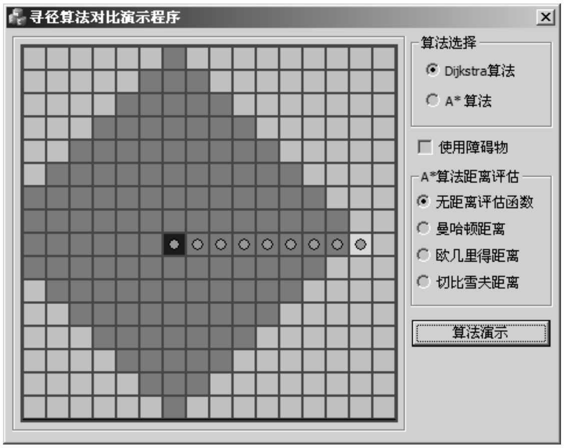


图 21-1 Dijkstra 算法搜索效率图示（无障碍物的情况）

那么，有障碍物的情况下 Dijkstra 算法的搜索效率如何呢？我们在地图中加入一段墙组成的障碍物（深灰色方块组成的 L 形障碍物），Dijkstra 算法的结果显示如图 21-2 所示，几乎搜遍了整个地图中的所有点，但是找到的路径确实就是最短路径。

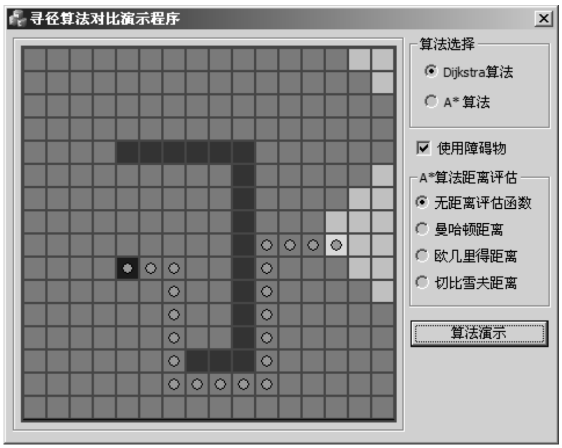


图 21-2 Dijkstra 算法搜索效率图示（有障碍物的情况）

21.3 带启发的搜索算法——A*算法

Dijkstra 算法从带权有向图的角度寻找图中顶点之间的最短路径，只是简单地做广度优先搜索，忽视了许多有用的信息。盲目搜索的效率很低，耗费很多时间和空间。考虑到实际地图上的两个点之间存在的位置和距离信息，是否可以利用这些信息得到一种高效的寻径算法呢？很显然，我们需要一种启发式搜索算法。一提到启发式搜索，首先要想到的就是启发函数，也就是评估函数。评估函数的作用就是根据起点和终点的位置和距离信息给出下一步需要搜索各个位置的评估值，启发式搜索算法可以有以下三种方式利用这些评估值。

- 根据评估结果，每次选择评估值最高的位置开始下一步搜索，避免盲目的穷举搜索；
- 决定搜索的顺序，按照评估值的高低排序，从评估值最高的位置开始下一步搜索（如果评估值高的位置没找到结果，则评估值较低的位置也能被顺序处理到）；
- 剪枝，去除一些明显不可能得到最优结果的搜索位置，提高搜索效率。

对盲目的穷举搜索来说，启发式搜索显然更高效，但是如果评估函数选择不当，也存在得不到正确结果的风险。

寻径算法中常见的启发式搜索算法有 BFS（Best-First Search）搜索算法和 A*算法。BFS 算法是在广度优先搜索算法的基础上加入评估函数，通过评估函数剪枝，避免一些明显不可能得到最短路径的搜索动作。在没有障碍物的地图上，其算法效果接近 A*算法，总体上远远优于 Dijkstra 算法。但是 BFS 算法因为评价函数只考虑位置方向信息，基于贪心策略，总是试图向最接近目标点的方向移动，使得这种算法在有障碍物的地图上表现不佳，很多情况下得到的路径都不是最短路径。图 21-3 就是用 A*算法模拟的 BFS 算法在有障碍物地图上得到的效果，由于过于强调与终点的距离，使得算法在启发函数的引导下一路向右，直到碰到障碍物才转向，这显然不是最短路径。

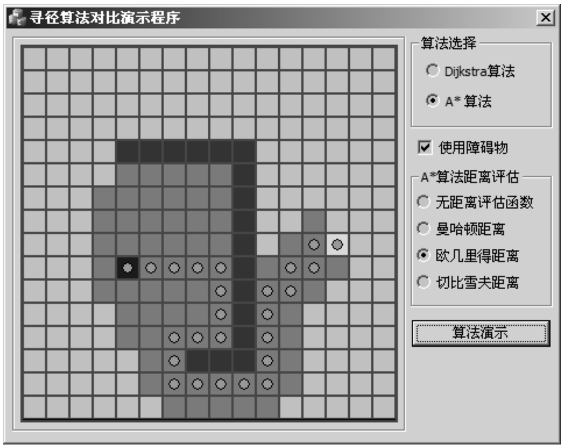


图 21-3 BFS 算法在有障碍物的情况下的表现

A*算法既能像 Dijkstra 算法那样搜索到最短路径，又能像 BFS 算法一样使用启发函数进行启发式搜索，是目前各种寻径算法中最受欢迎的选择。即使在有障碍物的情况下，选择合适的距离评估函数，A*算法基本上都能搜索到最短路径。本节我们将介绍 A*算法，其中启发函数的距离评估算法分别采用曼哈顿距离、欧氏几何平面距离和切比雪夫距离，寻径算法演示程序会以图示的方式给出它们的差别。

21.3.1 A*算法原理

BFS 算法在有障碍物的情况下会失败，原因在于其评估函数只考虑当前点与终点的距离，其策略是选择与终点最近的点进行搜索。Dijkstra 算法则只关注当前点与起点的距离，其策略是选择与起点最近的点开始搜索（与起点最近意味着从起点到当前点是最短路径，一旦当前点就是终点，那自然就是到终点的最短路径）。那么将二者结合起来会如何呢？这就是 A*算法的启发原理。

A*算法的启发函数采用的计算公式是：$F(n) = G(h) + H(n)$。$F(n)$就是 A*算法对每个点的评估函数，它包含以下两部分信息。

- ❑ $G(n)$ 是从起点到当前节点 n 的实际代价，也就是从起点到当前节点的移动距离。相邻的两个点的移动距离是 1，当前点距离起点越远，这个值就越大。
- ❑ $H(n)$ 是从当前节点 n 到终点的距离评估值。这是一个从当前节点到终点的移动距离的估计值。

在这个计算公式中，如果我们设 $G(n)$的值总是 0，则算法的效果类似于 BFS 算法。图 21-3 所示的结果就是将 A*算法中的 $G(n)$始终赋值为 0 得到的效果，与 BFS 算法的结果比较相似。反过来，如果设 $H(n)$的值总是 0，则算法可退化得到类似 Dijkstra 算法的效果，如图 21-4 所示。

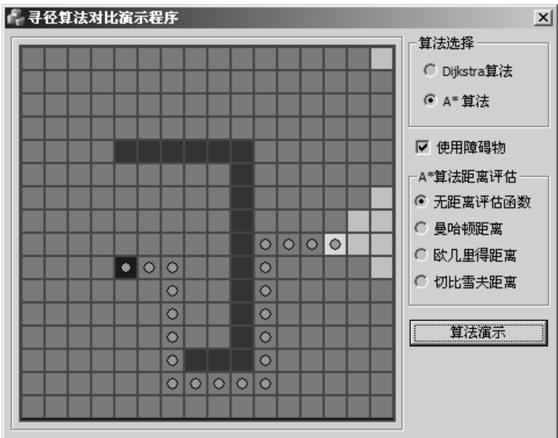


图 21-4 A*算法退化为 Dijkstra 算法的效果

A*算法的搜索过程需要两个表，一个是 OPEN 表，存放当前已经被发现但是还没有搜索过

的节点。另一个是 CLOSE 表，存放已经搜索过的节点。A*算法通过以下步骤搜索最短路径。

(1) 初始化 OPEN 表和 CLOSE 表，将起点加入到 OPEN 表中；

(2) 从 OPEN 表中取出当前 $F(n)$值最小的节点作为当前搜索节点 U，将 U 节点加入到 CLOSE 表中；

(3) 对于每一个与 U 可连通的节点（障碍物不相通）V，考察 V：

如果 V 已经在 CLOSE 表中，则对该节点不做任何处理；

如果 V 不在 OPEN 表中，则计算 $F(V)$，将 V 的前驱节点设置为 U 并将 V 加入到 OPEN 表中；

如果 V 在 OPEN 表中，比较 $G(U) + 1$ 与 $G(V)$的大小（$H(V)$的值是不变的），如果 $G(U) + 1 < G(V)$，则令 $G(V) = G(U) + 1$，同时将 V 的前驱节点设置为 U；

重复步骤(2)和(3)，直到第(2)步得到的搜索节点 U 就是终点为止，此时算法结束。

21.3.2 常用的距离评估函数

$H(n)$是 A*算法的距离估计值，A*算法需要一个距离评估函数来计算这个值。有很多距离评估函数可供选择，本节介绍曼哈顿距离、欧氏几何平面距离和切比雪夫距离。在没有障碍物的地图上，三种距离评估函数所得到的效果是一样的，如图 21-5 所示。但是在有障碍物的地图上，三种距离评估函数的效果略有差异。特别地，如果距离评估函数总是返回 0，则可令 A*算法退化为 Dijkstra 算法的效果，在图 21-4 中我们已经看到了这种效果。

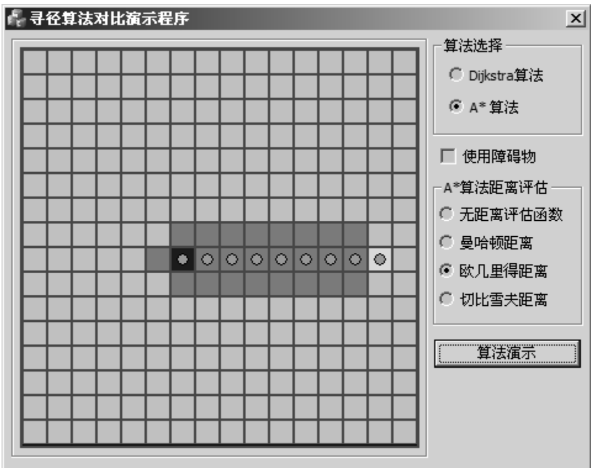


图 21-5 在没有障碍物的情况下，欧几里得距离评估函数的效果

1. 曼哈顿距离

曼哈顿距离（Manhattan distance）是 19 世纪的赫尔曼·闵可夫斯基所创的词语，曼哈顿距离的命名原因是从规划为方型建筑区块的城市（如曼哈顿）中寻找最短行车路径问题所引入的命

名，所以它又被称为出租车几何距离。从数学上描述曼哈顿距离是两个点在各个坐标轴上的距离差值的总和，n 维几何空间中的曼哈顿距离的数学描述为：

$$D_{\text{Manhattan}}(p,q)=\sum_{i=1}^{n}(px_i-qx_i)$$

对于二维平面上的两个点(x_1, y_1)和(x_2, y_2)，其曼哈顿距离就是：

$$D=\left|(x_1-x_2)\right|+\left|y_1-y_2\right|$$

即欧氏几何平面距离在直角坐标系中两个坐标轴上的投影的距离之和。本章介绍 A*算法用的曼哈顿距离实现代码是：

```
double ManhattanDistance(const ANODE& n1, const ANODE& n2)
{
    return (std::abs(n1.i - n2.i) + std::abs(n1.j - n2.j));
}
```

在有障碍物的地图上，使用曼哈顿距离评估函数的 A*算法效果如图 21-6 所示。

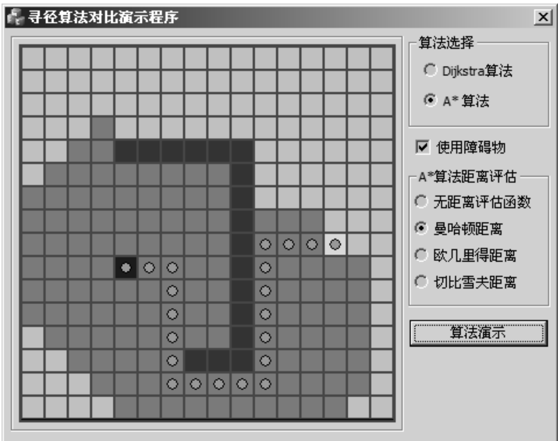


图 21-6 使用曼哈顿距离的 A*算法效果

2. 欧氏几何平面距离

欧氏几何平面距离（Euclidean distance）又称为欧氏距离或欧几里得距离，它的数学定义是 n 维空间中两个点之间的真实距离（几何距离），其数学符号可描述为：

$$D_{\text{Euclidean}}(p,q)=\sqrt{\sum_{i=1}^{n}(px_i-qx_i)^2}$$

对于二维平面上的两个点(x_1, y_1)和(x_2, y_2)，其欧氏几何平面距离就是：

$$D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

即平面几何中两点之间的几何距离。本章介绍 A*算法用的欧氏几何平面距离实现代码是：

```
double EuclideanDistance(const ANODE& n1, const ANODE& n2)
{
    return std::sqrt(double(n1.i - n2.i)*(n1.i - n2.i) + (n1.j - n2.j)*(n1.j - n2.j));
}
```

在有障碍物的地图上，使用欧氏几何平面距离评估函数的 A*算法效果如图 21-7 所示。可以看到与 Dijkstra 算法得到的最短路径稍有差别，这是因为欧氏几何平面距离强调对角线方向上的距离，但是我们的演示程序只在一个顶点的上下左右四个方向视为联通方向，对角线方向相邻的节点不是联通节点，所以最后一段变成了折线。后面我们将介绍距离评估函数 $H(n)$与 A*算法的关系。

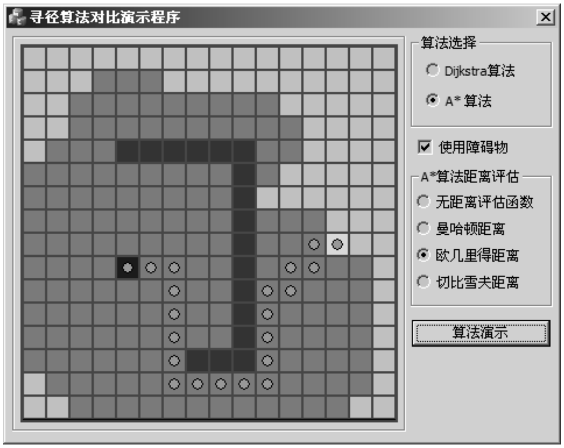


图 21-7 使用欧氏几何平面距离的 A*算法效果

3. 切比雪夫距离

切比雪夫距离（Chebyshev distance）是由一致范数（uniform norm）（或称为上确界范数）所衍生的度量，从数学上理解，对于两个向量 p 和 q，其切比雪夫距离就是向量中各个分量的差的绝对值中最大的那一个，用数学符号可描述为：

$$D_{\text{Chebyshev}}(p,q) = \max(|p_i - q_i|)$$

特别情况下，对于二维平面上的两个点(x_1, y_1)和(x_2, y_2)，其切比雪夫距离就是：

$$D = \max(|x_1 - x_2|, |y_1 - y_2|)$$

即两个点之间的切比雪夫距离就是两个方向上坐标数值差的最大值。本章介绍 A*算法用的切比雪夫距离实现代码是：

21

```
double ChebyshevDistance(const ANODE& n1, const ANODE& n2)
{
    return std::max<double>(std::abs(n1.i - n2.i), std::abs(n1.j - n2.j));
}
```

在有障碍物的地图上，使用欧氏几何平面距离评估函数的 A*算法效果如图 21-8 所示。

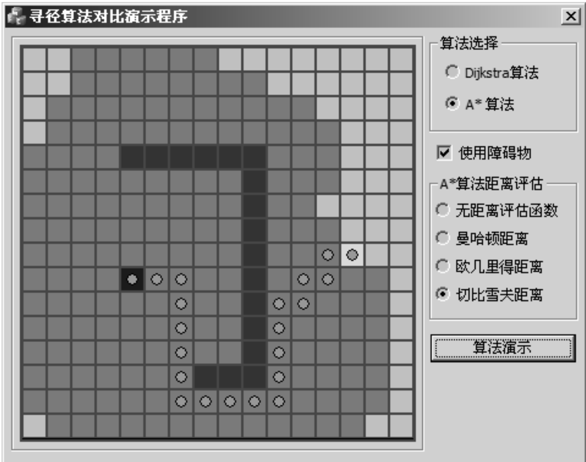


图 21-8 使用切比雪夫距离的 A*算法效果

4. *H*(*n*)与 A*算法的关系

距离评估函数 *H*(*n*)与 A*算法的结果之间存在很微妙的关系，前面介绍过，如果令 *H*(*n*)始终为 0，相当于一点启发信息都没有，则 A*算法退化为 Dijkstra 算法，这种情况被称为最差的 A*算法（尽管如此，可以确保得到最短路径）。*H*(*n*)的值越小，启发信息越少，搜索范围越大，速度越慢，但是越有希望得到最短路径。*H*(*n*)值越大，启发信息越多，搜索范围越小，速度越快，但是有可能得不到真正的最短路径。当 *H*(*n*)大到一定程度，*F*(*n*)公式中 *G*(*n*)的值可以被忽略，则 A*算法演化成 BFS 算法，速度最快，但是不一定能得到最短路径。

这是一个很有意思的关系，A*是一个很灵活的算法，通过调整 *G*(*n*)和 *H*(*n*)函数，可以使得 A*算法在速度和准确性之间获得一个折中的效果。在很多情况下，让游戏中的人物沿着一条近似最短的路径到达目的地就可以了，不一定要走最短路径。

21.3.3 A*算法实现

A*算法实现的关键是维护 OPEN 表和 CLOSE 表，其中对 OPEN 表的主要操作就是查询 *F*(*n*)最小的节点和删除节点，因此我们考虑在算法实现时将 OPEN 设计为有序表。STL 中的 `multiset` 天然具有排序特征，因此我们考虑使用 `std::multiset` 表达 OPEN 表。`AStar()`函数是 A*算法的实现，从注释可以看到与 21.3.1 节介绍的 A*算法三个步骤一一对应。`ExtractMiniFromOpen()`函数从 OPEN 表中取出 *F*(*n*)值最小的一个节点，OPEN 表已经根据 *F*(*n*)的值从低到高排序，因此

ExtractMiniFromOpen()函数所做的事情就是从 OPEN 表中取出第一个节点。

```
void AStar(ASTAR_GRAPH *graph, GRID_CELL *gc)
{
    //步骤(1)
    graph->open.insert(graph->source);

    //步骤(2)
    ANODE cur_node;
    while(ExtractMiniFromOpen(graph, cur_node))
    {
        graph->close.push_back(cur_node);
        if(cur_node == graph->target)
        {
            UpdateCellInfo(graph, gc);
            break;
        }

        //步骤(3)
        for(int d = 0; d < COUNT_OF(dir); d++)
        {
            ANODE nn = {cur_node.i + dir[d].y, cur_node.j + dir[d].x, 0, 0};
            if((nn.i >=0) && (nn.i < N_SCALE) && (nn.j >=0) && (nn.j < N_SCALE)
                && (gc->cell[nn.i][nn.j].type != CELL_WALL)
                && !IsNodeExistInClose(graph->close, nn.i, nn.j))
            {
                std::multiset<ANODE, compare>::iterator it;
                it = find(graph->open.begin(), graph->open.end(), nn);
                if(it == graph->open.end()) /*nn 不在 open 列表*/
                {
                    nn.g = cur_node.g + 1; //将 g 始终赋值为可得到 BFS 算法的效果
                    nn.h = ManhattanDistance(nn, graph->target);
                    nn.prev_i = cur_node.i;
                    nn.prev_j = cur_node.j;
                    graph->open.insert(nn);
                    gc->cell[nn.i][nn.j].processed = true;
                }
                else  /*nn 在 open 列表中*/
                {
                    if((cur_node.g + 1.0) < it->g)
                    {
                        it->g = cur_node.g + 1.0;
                        it->prev_i = cur_node.i;
                        it->prev_j = cur_node.j;
                    }
                }
            }
        }
    }
}
```

这就是 A*算法的实现，并不像它的名字那么神秘，正如广告说的那样：简约而不简单。A*算法是各种游戏中最常用也是最好的寻径算法之一。

21.4 总结

从 Dijkstra 算法和 A*算法的实现可知，它们都是很简单的算法。Dijkstra 算法的时间复杂度是 $O(n^2)$，其中 n 是有向图中顶点的个数。对于不含负权边的有向图来说，Dijkstra 算法是目前最快的单源最短路径算法。如果有向图中含有负权的边，则可以使用 Floyd-Warshall 算法求解最短路径。Floyd-Warshall 算法可以求解有向图中任意两点之间的最短距离，其时间复杂度是 $O(n^3)$。

A*算法兼有 Dijkstra 算法和 BFS 算法的特点，在速度和准确性之间具有很大的灵活性。除了调整 $G(n)$和 $H(n)$获得不同效果，A*算法还有很多提高效率的改进算法。比如在地图很大的情况下，可以使用二叉堆来维护 OPEN 表以获得更好的效率。对于环境和权重都不断发生变化的动态网络，还有动态 A*算法（又称 D*算法）。

Dijkstra 算法在地图和导航软件中得到了广泛的应用，A*算法在游戏软件中也得到了广泛的应用，它们都是很简单的算法，但是却得到了广泛的应用，这也是小算法解决大问题的现实例子。

21.5 参考资料

[1] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[2] 维基百科：http://zh.wikipedia.org/zh-cn/Dijkstra 算法

[3] 维基百科：http://zh.wikipedia.org/wiki/A*搜寻算法

[4] Amit's A* Pages：http://theory.stanford.edu/~amitp/GameProgramming/

[5] A* Pathfinding for Beginners：http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/a-pathfinding-for-beginners-r2003

[6] Smart Moves: Intelligent Path Finding：http://www.gamasutra.com/view/feature/3215/myths_ and_facts_in_avoiding_.php

第 22 章 俄罗斯方块游戏

俄罗斯方块游戏（Tetris）是前苏联科学家阿列克谢·帕基特诺夫在 1984 年 6 月利用空闲时间所编写的一个游戏程序（如图 22-1 所示）。帕基特诺夫当时是俄罗斯科学院计算机中心工作的数学家，据说他编写这个游戏最初的目的是用来测试一种新型计算机的性能。至于这个游戏名字的来历，据说是因为帕基特诺夫喜欢网球（Tennis）运动，于是他把来源于希腊语的 tetra（意为“四”）与其结合，造了“tetris”一词，不过这个说法也未经证实。从 1988 年开始，俄罗斯方块游戏风靡全世界。从最初的街机和掌上游戏机到计算机平台，再到现在的手机、平板等移动平台，它深受全世界游戏迷的喜爱。

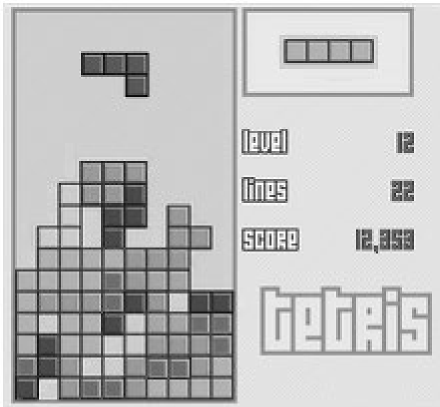


图 22-1　俄罗斯方块游戏

编写一个俄罗斯方块游戏，涉及键盘控制、定时器、UI 和复杂数据结构定义和使用，非常具有挑战性，很多编程爱好者都自己编写过俄罗斯方块游戏。但是大家有没有想过，是否可以脱离人的控制，让计算机自己玩俄罗斯方块游戏呢？事实上，很多高级的俄罗斯方块游戏都提供了电脑演示或电脑提示的功能，那么，如何让计算机知道把各种形状的板块放在最合适的位置上呢？本章我们就来介绍一种简单的人工智能（AI）算法，让计算机玩俄罗斯方块游戏。这类简单的人工智能的本质就是通过评估函数，对一个局面以及局面的演化结果进行评估，选择较好的一个局面作为演化结果。类似的算法还被用在棋类游戏中，第 23 章将会介绍此类人工智能算法在棋类游戏中的应用。

22.1　俄罗斯方块游戏规则

俄罗斯方块游戏一共有七种形状不同的板块，每种板块都由四个小方块组成，如图 22-2 所示，这些板块被冠以一个大写英文字母作为名字，分别是：I、J、L、O、S、T 和 Z。游戏的区

域是一个宽度为 10 个小方块、高度为 20 个小方块的长方形区域。在游戏进行的过程中，不同形状的板块从游戏区域上方随机落下，在板块下落的过程中，玩家可以通过游戏机的控制按钮，以 90 度为单位顺时针或逆时针旋转板块，同时也可以左右移动板块。当一个板块下落到游戏区域最下方或着落到其他板块上无法再向下移动时，就会固定在该处，然后新的板块从游戏区域的上方开始落下。当区域中某一横行的小方格全部由方块填满时，则该行会被消除并成为玩家的得分。同时消除的行数越多，得分也越多，比如消除一行得 100 分，同时消除两行可以得 200 分，同时消除三行可以得 400 分，同时消除四行可以得 800 分。没有被消除掉的方块不断堆积起来，一旦堆到游戏区域顶端，玩家便告输，游戏结束。

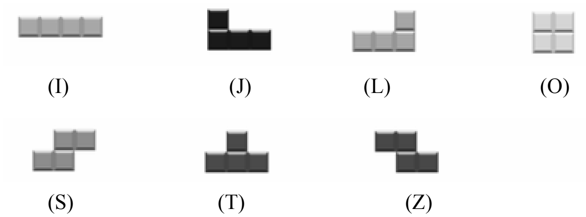


图 22-2　俄罗斯方块形状示意图

一般来说，游戏还会提示下一个将要落下的板块的形状，熟练的玩家会利用下一个板块的形状评估现在要如何摆放当前的板块。玩家玩俄罗斯方块游戏的目的是得更高的分数和玩更长的时间，但是游戏能不断进行下去对商用游戏不太合适，所以一般俄罗斯方块游戏的程序设计都会随着游戏的进行而提高难度，比如加快板块的下落速度，随机增加一些带空格的行等。

俄罗斯方块游戏在传播过程中，出现了很多有意思的改版，有的是增加了更多形状的板块，有的是增加一种能穿透固定方块的单格小方块，使得玩家能够有机会“补上”游戏区域下方的“空洞”。还有 2.5 维和 3 维俄罗斯方块游戏，以及利用整面墙上的窗户模拟俄罗斯方块游戏的有趣试验，大家可以在本章的参考资料[5]中了解到这些内容。本章介绍的智能算法都是基于标准俄罗斯方块游戏的规则设计的，可能不适用于各种改版游戏规则，但是作为一种基础方法，读者可以在此基础上增加对其他规则的支持。

22.2　俄罗斯方块游戏人工智能的算法原理

在探讨计算机的俄罗斯方块游戏智能算法之前，我们先研究一下人类玩家玩这个游戏的一些基本策略。玩家玩这个游戏，首先要能够玩尽量长的时间，这就要求要尽可能地消除行，避免累积高度太高。其次是尽量多的得分，利用规则消除加分的特点，尽量一次消除多行。在遇到板块形状很难处理的情况，要选择产生空格子少的摆放方法，尽量避免出现“空洞”。很多情况下，当一个板块可以摆放在多个位置的时候，玩家需要根据自己的经验选择一个对下一步操作最有利的位置摆放这个板块，这就涉及局面评估的问题。

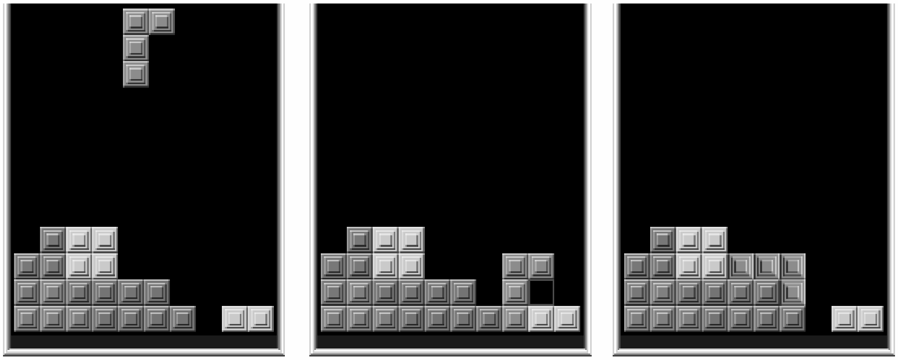
玩家可以根据自己的经验迅速做出判断，但是计算机做不到这一点。使用神经网络+专家系统可以使计算机具有一点“经验”，但是对于俄罗斯方块这样的游戏有点大材小用。对于此类问题，通常的做法是设计一个局面评价函数，对各种可能的局面进行评估，根据评估结果选出最好的一个局面进行实施。对于俄罗斯方块游戏而言，我们把 10×20 个小格子组成的游戏区域称作一个“棋盘”，所谓的局面就是一个板块摆放在某个位置后这个“棋盘”的状态。如果一个板块有多个位置可以摆放，就会产生多个“棋盘”状态，使用评价函数对每个状态进行评估，根据评估的结果将板块摆放在最佳位置上。

22.2.1 影响评价结果的因素

影响评价结果的因素是多方面的，对这些因素需要一个统一的考虑，选择一个合理的评估策略。每一个板块放在什么位置，都会造成一系列的状态参数变化，根据俄罗斯方块游戏的规则，我们整理出相关的参数有如下几个。

- 当一个板块摆放后，与这个板块相接触的小方块的数量是一个需要考虑的参数。很显然，与之接触的小方块越多，说明这个板块摆放在该位置后产生的空格或“空洞”越少，如果一个“棋盘”局面中空的小方格或“空洞”数量少则说明这个局面对玩家有利。
- 当一个板块摆放在某个位置后，这个板块最高点的高度是一个需要考虑的参数。这个高度会影响整体的高度，当有两个位置可选则摆放板块时，应该优先选择放置在板块最高点的高度比较低的位置上。
- 当一个板块摆放在某个位置后能消除的行数是一个重要参数。毫无疑问，能消除的行越多越好。
- 游戏区域中已经被下落板块填充的区域中空的小方格的数量也是评价游戏局面的一个重要参数。很显然，每一行中的空小方格数量越多，局面对玩家越不利。
- 游戏区域中已经被下落板块填充的区域中“空洞”的数量也是一个重要参数。如果一个空的小方格上方被其他板块的小方格挡住，则这个小方格就形成“空洞”。“空洞”是俄罗斯方块游戏中最难处理的情况，必需等上层的小方块都被消除后才有可能填充“空洞”。很显然，这是一个能恶化局面的参数。

简单地理解，摆放一个板块的策略就是：板块放置的位置越靠下越好，方块之间越紧密越好，能消除的（行）方块数量越多越好。当然，这些参数要统一考虑，片面地突出某一方面的参数，会起到物极必反的作用。举个例子，选择摆放板块的位置时，如果能消除一行当然是非常理想的位置，但是如果评估函数过分重视消除参数的影响，反而会导致一些非常不利的局面出现。图 22-3 就展示了一种单方面突出消除参数而导致糟糕局面的情况。在图 22-3a 所示的局面上摆放板块 J，如果突出消除参数的因素，评价函数最终的结果可能会选择图 22-3b 所示的放置位置，因为这样能消除一行。但是这样摆放的结果是出现了“空洞”，这是俄罗斯方块游戏中最棘手的情况。事实上，在这种局面下，对玩家最有利的摆放方法是放在图 22-3c 所示的位置上。



(a)　　(b)　　(c)

图 22-3　过分突出消除参数而导致不利局面的示例

22.2.2　常用的俄罗斯方块游戏人工智能算法

可以说，评估函数的优劣决定了游戏智能的强弱，这么多参数中，如何给出一种均衡的评估策略，使得评估函数总是能够给出最有利的评价结果？这个问题不好回答，首先要根据问题的本质确定这些参数和评估结果是线性关系还是各种非线性关系，其次是确定各种参数在评估结果曲线上的系数。确定这些系数没有好的方法，如果参数不复杂，而且是简单的线性关系，可以通过多次对比实践逐步调整这些参数的系数。如果参数复杂且参数之间的关系复杂，多数人会选择使用神经网络之类的学习算法，利用大量的游戏局面数据进行“训练”，最终收敛出一组能接近最优结果的系数。但是这种方法也存在随机性比较大的问题，受“训练”数据的影响比较大，如果“训练”数据不够多，得到的结果会非常差。

幸运的是，我们不需要做这些棘手的事情，这个领域的先行者们为我们留下了他们的经验和研究结果。最初人们热衷于研究俄罗斯方块游戏的“不死”算法，也有一些人开始研究怎么打败俄罗斯方块游戏的 AI 程序。1997 年，Heidi Burgiel 在参考资料[6]中证明了完全随机的俄罗斯方块游戏最终一定会结束，于是人们把热情转移到如何让程序的 AI 能够获得更高的积分或消除更多的行（平均值）。在这个过程中出现了很多著名的 AI 算法，比如 Pierre Dellacherie 算法、Colin Fahey 算法、Roger LLima/Laurent Bercot/Sebastien Blondeel 算法、James & Glen 算法和 Thiery & Scherrer 算法等。Colin Fahey 算法和 James & Glen 算法都支持 two-piece 算法，所谓的 two-piece 算法，就是在评估的过程中将当前板块形状和下一块板块形状一起进行评估和计算。Colin Fahey 在自己的网站上公开了算法的实现代码，同时还发布了一个算法模拟平台，各种俄罗斯方块游戏的 AI 算法可以在这个平台上进行对比和评估。大家可以通过参考资料[5]中给出的链接下载 Colin Fahey 的实现代码和这个模拟平台。在 2003 年之前，Colin Fahey 算法在这个模拟器平台上取得了非常好的效果。

在评估过程中只考虑当前板块形状的 one-piece 算法相对简单一些，但是取得的效果一点也

不比 two-piece 算法差，甚至要强于 two-piece 算法。Pierre Dellacherie 算法和 Thiery & Scherrer 算法都是比较著名的 one-piece 算法，当然，James & Glen 算法有 one-piece 算法的版本。Colin Fahey 算法如果屏蔽对下一板块的判断，也可以当成 one-piece 算法，但是由于 Colin Fahey 算法是针对 two-piece 的情况研究的算法，在 one-piece 的情况下性能很差。Christophe Thiery 和 Bruno Scherrer 在参考资料[7]中介绍了这些算法的评估原理和评分方法。当然，Bruno Scherrer 和 Christophe Thiery 两人也发布了 Thiery & Scherrer 算法的实现（毕竟这就是他们俩研究的算法嘛），据说可以平均消除 3500 万行。2009 年他们发布了这个算法的 1.4 版，立即超越 Pierre Dellacherie 算法，成为当年 one-piece 算法的 No.1，大家可以通过参考资料[8]提供的链接下载他们的算法实现。

本章我们将重点介绍相对简单一点的 Pierre Dellacherie 算法，该算法每次只考虑当前板块的情况，是一种 one-piece 算法。Pierre Dellacherie 算法虽然简单，但是性能一点都不弱，Colin Fahey 在他的网站上非常推崇 Pierre Dellacherie 算法，称其是 one-piece 算法中最好的算法（2003 年）。Pierre Dellacherie 算法最好的结果是能消除 200 多万行，平均也能达到 65 万行。在 2003 年，Pierre Dellacherie 算法是 one-piece 算法中公认的 No.1。

22.2.3 Pierre Dellacherie 评估算法

22.2.1 节介绍了一些评价俄罗斯方块游戏局面的参数，但是这些参数都是一些抽象的参数，如何具体使用这些参数进行评估计算呢？对于这些参数，不同的算法有不同的使用策略，本节要介绍的 Pierre Dellacherie 评估算法就是其中一种评价策略。2003 年，法国人 Pierre Dellacherie 在 Colin Fahey 的平台上提交了一种 one-piece 算法，该算法的结果超过了 Colin Fahey 算法，取得了平均消除 65 万行的成绩，成为 2003 年智能程度最高的 one-piece 人工智能算法。

Pierre Dellacherie 算法将 22.2.1 节介绍的影响俄罗斯方块游戏的抽象参数转化为 6 种具体的属性，并详细定义了这 6 种属性。

- `landingHeight`：指当前板块放置后，板块重点距离游戏区域底部的距离（以小方格为单位）；
- `erodedPieceCellsMetric`：这是消除参数的体现，它代表的是消去的行数与当前摆放的板块中被消去的小方格数的乘积；
- `boardRowTransitions`：如果把每一行中的小方格从有小方块填充到无小方块，或从无小方块到有小方块填充视作一次“变换”的话，这个属性就是各行中发生变换的次数之和；
- `boardColTransitions`：关于“变换”的定义和 `boardRowTransitions` 一样，只是以列为单位统计变换的次数；
- `boardBuriedHoles`：各列中“空洞”的小方格数之和；
- `boardWells`：各列中“井”的深度连加之和。

`landingHeight` 属性比较简单，无需多做说明。`erodedPieceCellsMetric` 属性体现了消除参数的影响，但是对它进行了适当的折中。每个板块由四个小方块组成，如果能同时将这个板块的四个小方块都消除，其结果就是“行数 × 4”，将取得明显优势。但是如果只能消除一个小方块，其结

果就是 1，所产生的影响很容易被形成一个“空洞”或引起一次“变换”所抵消，这样就可以避免类似图 22-3 所示的例子中的那种“偏激”的选择。boardRowTransitions 和 boardColTransitions 属性反映的是小方块摆放的紧密程度。这个也比较容易理解，小方块摆放得越紧密，其间的空格就越少，小方格状态之间的变换就越少。但是需要注意一点，这种变换要考虑边界因素。可以这样理解，如果紧邻边界的行或列是空小方格，则视为一次“变换”，即边界作为被填充的小方格参与计算。

boardBuriedHoles 是一列中“空洞”的小方格数量之和。所谓的“空洞”就是某一列中顶端被小方块填堵住的空小方格，如图 22-4 所示，带有方框标识的就是“空洞”。形成空洞是俄罗斯方块游戏中最坏的局面，要极力避免这种情况，因此 Pierre Dellacherie 算法给“空洞”的系数是 –4。boardWells 是“井”的深度连加之和。首先来定义什么是“井”，“井”就是两边（包括边界）都由方块填充的空列。图 22-5 就是很多资料上常引用的“井”的示意图，其中带有方框标识的就是两个“井”。“井”的评价记分采用的是连加求和，一个“井”中连续的空小方格有 1 个就计 1，有两个就计 1+2=3，有三个就计 1+2+3=6，以此类推。如 22-5 中两个“井”的记分之和就是 (1+2)+(1+2+3)= 9。

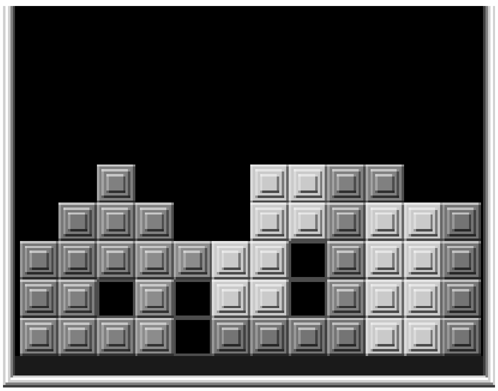


图 22-4　“空洞”示意图

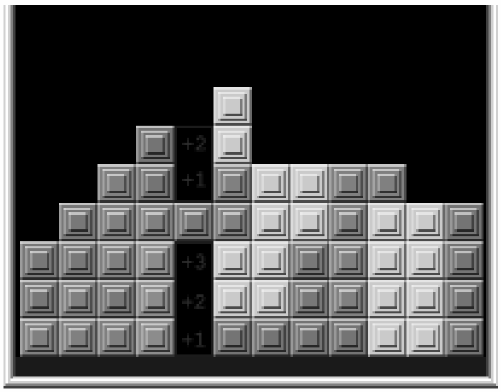


图 22-5　“井”示意图

接下来介绍 Pierre Dellacherie 算法的评估函数。该评估函数以上述 6 个属性为输入参数，采用线性组合的方式，计算出最后的评估值（value），其计算方法如下：

$$\text{value} = -\text{landingHeight} + \text{erodedPieceCellsMetric} - \text{boardRowTransitions} - \text{boardColTransitions} - (4 * \text{boardBuriedHoles}) - \text{boardWells} \tag{22-1}$$

对每个局面计算应用上述公式计算 value 值，取最大的一个作为最后的选择。如果两个局面的评分相同怎么办？两个局面的 value 值相同是一种很普遍的情况，为此 Pierre Dellacherie 算法又定义了一个优先度的概念，当两个局面的 value 值相同的时候，取优先度大的那个作为最后的选择，优先度的定义如下。

如果板块摆放在游戏区域的左侧（1 ~ 5 列）：

priority = 100 × 板块需要水平平移次数 + 10 + 板块需要旋转的次数

如果板块摆放在游戏区域的右侧（6 ~ 10 列）：

priority = 100 × 板块需要水平平移次数 + 板块需要旋转的次数

假如游戏中新的板块总是从游戏区域的中间开始落下，那么“板块需要水平平移次数”就是将板块摆放在所选位置时需要水平移动多少个小方格。每个板块最终摆放在指定位置后，其形态不一定就是初始形态，可能需要做一些旋转操作才能以此形态放置，这些旋转操作的次数就是“板块需要旋转的次数”。

以上就是 Pierre Dellacherie 评估算法的核心内容，主要就是式(22-1)所代表的评估函数，这个决定了俄罗斯方块游戏 AI 的智能。接下来，我们就以 Pierre Dellacherie 评估算法为基础，编写一个自动玩俄罗斯方块游戏的程序。

22.3 Pierre Dellacherie 算法实现

Demaine、Hohenberger 和 Liben-Nowell 在参考资料[3]中初步论证了俄罗斯方块游戏是 NP 完全问题（NP-complete），这使得所有人都彻底放弃了寻找俄罗斯方块游戏的数学公式解法。目前主要的几种俄罗斯方块游戏人工智能算法，都采用了穷举算法，只是在穷举实现的细节上稍有不同。因此，基于传统俄罗斯方块游戏规则制作一个 one-piece 算法相当地简单。总的来说，俄罗斯方块游戏的人工智能算法都由两个核心部分组成，其一是板块摆放动作引擎，此引擎负责产生各种板块的摆放方法；其二是评估函数，对每种板块摆放方法进行评估。

板块摆放动作引擎就是穷举所有可能的板块摆放方法，对于板块的所有可能的旋转状态，从左到右依次进行尝试。这项工作的核心是设计好数据结构，处理好板块之间的冲突检测。评估函数就使用 Pierre Dellacherie 评估算法，22.2.3 节已经介绍了这个评估算法的原理，只要将其实现算法写出来就算完成了。作为一个算法验证程序，不需要设计复杂的图形界面，结果评估和板块的摆放都是内存中的数据，可以使用简单的控制台界面将其展示出来，如图 22-6 所示。剩下的工作就是随机生成数千到数十万个板块，让我们的算法一一摆放它们，看看我们的算法能坚持多长时间或得多少分。

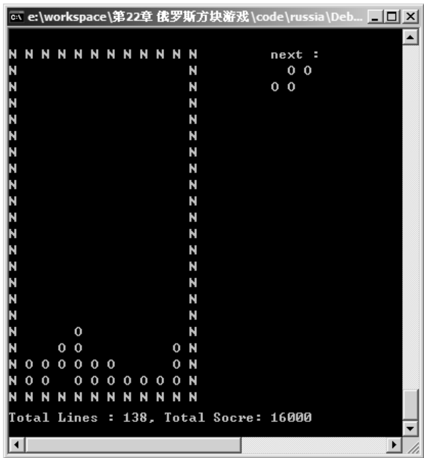


图 22-6 演示程序的输出界面

22.3.1 基本数学模型和数据结构定义

要让计算机理解俄罗斯方块游戏，需要定义数学模型。俄罗斯方块游戏的数学模型可采用棋盘类游戏常用的数学模型来定义，把游戏区域比拟为棋盘，用二维数组表示这个区域的状态，同时注意处理好边界问题。在此基础上，继续定义板块形状的数据结构，确定板块旋转和移动的数据定义以及板块冲突检测的规则。

1. 游戏和游戏区域

俄罗斯方块游戏包含几个关键要素，分别是当前游戏区域的状态、当前消除的行数、当前得分、下一个板块的形状以及 AI 算法。游戏区域有 10 × 20 个小方格组成，数据结构定义依然采用二维数组，因为要考虑边界的情况，所以定义为 12 × 22 的二维数组。我们用 1 表示小方格被占用的状态，用 0 表示小方格处于空的状态，用一个大于 0 的值表示边界方格，这是此类算法处理的常用技巧。当前已经消除的行数和当前得分是游戏进行过程中的两个状态，用整数分别表示它们就可以了。除此之外，增加一个表示当前最高行所在位置的标识：top_row，进行板块摆放位置穷举的时候，根据 top_row 指示的位置可以减少一些无谓的摆放尝试。

最后，RUSSIA_GAME 数据结构的定义如下：

```
typedef struct tagRussiaGame
{
    int board[BOARD_ROW][BOARD_COL];
    int top_row;
    int score;
    int lines;
}RUSSIA_GAME;
```

2. 板块形状的定义

标准俄罗斯方块游戏一共定义了 7 种板块形状，每种形状都由 4 个小方块组成，我们用一个 4 × 4 的小方格矩阵描述每一个板块形状，如图 22-7 所示。每种形状的板块通过旋转可以产生几种不同的形态，O 型板块不管如何旋转都只有一种形态。I、S 和 Z 型板块通过旋转可以产生两种形态，L、J 和 T 型板块通过旋转可以产生 4 种形态。图 22-7 显示了如何用 4 × 4 的矩阵描述这些形状经过旋转产生的各种形态，其中灰色显示的小方块表示板块的形状。由小方块组成的 4 × 4 矩阵一般用二维数组定义，板块的旋转可以采用两种策略。一种策略是给矩阵中的每个小方块设定一个坐标，需要旋转板块时就根据旋转的方向和角度重新计算每个小方块的坐标，使灰色显示的小方块变换到正确的位置上。另一种策略就是将每个板块旋转后可能产生的各种形态事先准备好，存放在一些列 4 × 4 矩阵中，需要旋转板块时就根据板块形状和旋转角度直接选择这些事先准备好的小方格矩阵使用。由于每种板块旋转所产生的形态是有限的，多则 4 种形态，少则 1 种形态，因此采用第二种策略并不会带来太大的存储负担，但是却可以大大简化算法实现，因此大多数俄罗斯方块游戏都是采用第二种策略来处理板块旋转。

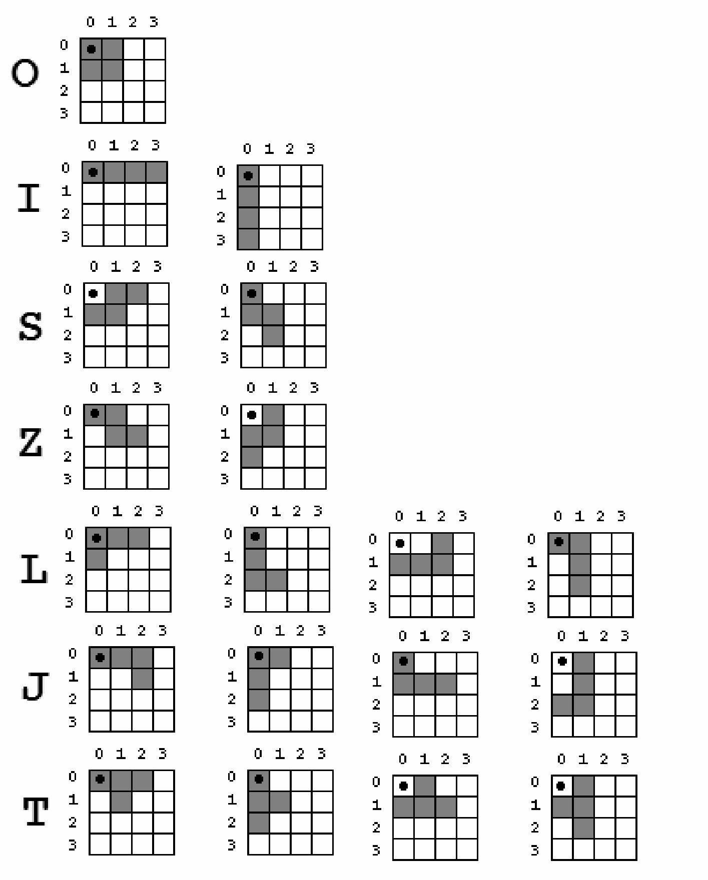


图 22-7 板块形状数据定义示意图

R_SHAPE 数据结构定义和存储一个板块的所有形态信息，r_count 是这个板块旋转时可能产生的不同形态的个数，对于 O 型板块来说，r_count 就是 1，对于 T 型板块来说，r_count 就是 4。shape_r 是一个最大长度为 4 的数组，存放 r_count 对应的每一种旋转形态。

```
typedef struct tagRShape
{
    B_SHAPE shape_r[MAX_SHAPE_R];
    int r_count;
}R_SHAPE;
```

B_SHAPE 是每种具体板块形态的 4×4 矩阵定义，二维数组 shape 就存储这个矩阵。shape 中的值如果是 1，则表示对应的小方格是板块形态的有效格子（对应图 22-7 中灰色显示的小方格），0 表示对应的小方格是无效的空格子，计算碰撞和摆放时可以忽略 0 对应的无效格子。width 和 height 定义板块形态在 4×4 矩阵中实际占用的宽度和高度，以图 22-7 所示的 S 型板块为例，其第 1 种旋转形态的宽度是 3，高度是 2，第二种旋转形态的宽度是 2，高度是 3。做碰撞检测计算

时，直接利用这两个值可以提高检测计算的效率。此外，在穷举板块的摆放位置时，也可以根据这两个值直接排除掉一些明显不合适的位置。

```
typedef struct tagBShape
{
    int shape[SHAPE_BOX][SHAPE_BOX];
    int width;
    int height;
}B_SHAPE;
```

根据以上数据结构定义，用 7 个 R_SHAPE 类型元素组成的数组存放事先准备好的板块旋转形态数据，算法实现过程中需要引用这些数据时，首先根据板块的形状编号从 R_SHAPE 数组中找到板块对应的 R_SHAPE 数据，然后就可以根据旋转角度找到 R_SHAPE 数据中对应的 B_SHAPE 数据。如果要穷举 R_SHAPE 板块的所有旋转状态，只需遍历 shape_r 数组即可。

22.3.2 算法实现

编写一个自动玩俄罗斯方块游戏的 AI 算法其实非常简单，就是要做两件事情：一是穷举板块的所有摆放形态和摆放位置，二是用评估函数对每种摆放方法进行评估，根据评估结果选择一个最佳摆放位置。22.3.1 节给出具体的的数据结构定义以后，原来抽象的方法描述和算法原理就可以用具体的方式实现了。首先来看看 Pierre Dellacherie 算法的评估函数如何实现。

1. Pierre Dellacherie 算法评估函数

Pierre Dellacherie 算法的评估函数包含 6 个属性，现在就来介绍如何从一个游戏“棋盘”局面中统计出这 6 个属性。首先是 landingHeight，这个非常简单，由于数组的下标 row 与高度是反对称的，需要做个取反计算：

```
int GetLandingHeight(RUSSIA_GAME *game, B_SHAPE *bs, int row, int col)
{
    return (GAME_ROW - row);
}
```

接下来计算 erodedPieceCellsMetric。GetErodedPieceCellsMetric 的算法实现也很简单，就是从 top_row 开始遍历所有的行，如果发现某一行可以消除，则计算当前板块形状中有多少小方块属于这一行。erodedRow 记录可以消除多少行，erodedShape 记录消除的行中有多少小方块是属于当前摆放的板块，最后返回它们的乘积。

```
int GetErodedPieceCellsMetric(RUSSIA_GAME *game, B_SHAPE *bs, int row, int col)
{
    int erodedRow = 0;
    int erodedShape = 0;
    int i = game->top_row;
    while(i < GAME_ROW)
    {
        if(IsFullRowStatus(game, i))
        {
            erodedRow++;
```

```
            if((i >= row) && (i <= (row + bs->height)))
            {
                int sline = i - row;
                for(int j = 0; j < bs->width; j++)
                {
                    if(bs->shape[sline][j] != 0)
                    {
                        erodedShape++;
                    }
                }
            }
        }
        i++;
    }

    return (erodedRow * erodedShape);
}
```

boardRowTransitions 和 boardColTransitions 的计算也非常简单。以 GetBoardRowTransitions() 函数的计算为例，从 top_row 开始遍历所有的行，对每一行统计“变换”。统计从左边界开始到右边界结束，注意这个算法里列下标是从 0 开始的，因为要从 board 区域中的边界开始计算。计算 boardColTransitions 的算法实现与 GetBoardRowTransitions()函数类似，指示将遍历方法从按照行遍历改成按照列遍历。

```
int GetBoardRowTransitions(RUSSIA_GAME *game, B_SHAPE *bs, int row, int col)
{
    int transitions = 0;
    for(int i = game->top_row; i < GAME_ROW; i++)
    {
        for(int j = 0; j < (BOARD_COL - 1); j++)
        {
            if((game->board[i + 1][j] != 0)&&(game->board[i + 1][j + 1] == 0))
            {
                transitions++;
            }
            if((game->board[i + 1][j] == 0)&&(game->board[i + 1][j + 1] != 0))
            {
                transitions++;
            }
        }
    }

    return transitions;
}
```


“空洞”是一个很关键的属性，但是计算 boardBuriedHoles 的算法并不复杂。遍历 board 的每一列，对每一列从 top_row 开始找第一个填充的小方块（第一个 while 循环），找到之后再继续找这个小方块之下所有的空小方格，统计它们的数量之和（第二个 while 循环）。

```
int GetBoardBuriedHoles(RUSSIA_GAME *game, B_SHAPE *bs, int row, int col)
{
    int holes = 0;
```

```
    for(int j = 0; j < GAME_COL; j++)
    {
        int i = game->top_row;
        while((game->board[i + 1][j + 1] == 0) && (i < GAME_ROW))
            i++;
        while(i < GAME_ROW)
        {
            if(game->board[i + 1][j + 1] == 0)
            {
                holes++;
            }
            i++;
        }
    }

    return holes;
}
```

“井”的计算仍然以列为单位进行扫描，对每一列从 top_row 开始处理，如果某个小方格是空状态，但是其左右相邻的两列（包括边界）都是填充的小方格，则统计井深的 wells 计数器+1。当遇到一个小方块是填充状态时，一个井深的统计结束，根据 wells 计数器计算 sum，然后 wells 计数器清 0，准备继续统计下一个“井”的深度。

```
int GetBoardWells(RUSSIA_GAME *game, B_SHAPE *bs, int row, int col)
{
    int wells = 0;
    int sum = 0;
    for(int j = 0; j < GAME_COL; j++)
    {
        for(int i = game->top_row; i <= GAME_ROW; i++)
        {
            if(game->board[i + 1][j + 1] == 0)
            {
                if((game->board[i + 1][j]!= 0)&&(game->board[i + 1][j + 2]!=0))
                {
                    wells++;
                }
            }
            else
            {
                sum += sum_n[wells];
                wells = 0;
            }
        }
    }

    return sum;
}
```

统计 sum 的时候，假如井深是 n，需要计算从 1 到 n 的和，这一步我们再次使用了以空间换时间的策略，预先计算好从 1 到 n 各数列的和，存放在 sum_n 表中，然后用 n 作为数组下标直接得到对应的和。游戏区域最高就是 20 行，因此井深不会超过 20，只需计算 20 个数列和存放在

sum_n 表即可。

```
int sum_n[] = { 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171, 190, 210 };
```

现在我们已经有了 6 个属性的计算方法，按照式(22-1)给出的计算方法写出评估函数即可：

```
int EvaluateFunction(RUSSIA_GAME *game, B_SHAPE *bs, int row, int col)
{
    int evalue = 0;

    int lh = GetLandingHeight(game, bs, row, col);
    int epcm = GetErodedPieceCellsMetric(game, bs, row, col);
    int brt = GetBoardRowTransitions(game, bs, row, col);
    int bct = GetBoardColTransitions(game, bs, row, col);
    int bbh = GetBoardBuriedHoles(game, bs, row, col);
    int bw = GetBoardWells(game, bs, row, col);

    evalue = (-1) * lh + epcm - brt - bct - (4 * bbh) - bw;

    return evalue;
}
```

最后是优先度选择，假如两个局面的评估值一样，就需要按照优先度进行选择，优先度计算的算法如下：

```
int PrioritySelection(RUSSIA_GAME *game, int r_index, int row, int col)
{
    int priority = 0;

    if(col < (GAME_COL / 2))
    {
        priority = 100 * ((GAME_COL / 2 - 1) - col) + 10 + r_index;
    }
    else
    {
        priority = 100 * (col - (GAME_COL / 2)) + r_index;
    }

    return priority;
}
```

这个算法实现基本上就是按照 22.2.3 节给出的公式进行计算，r_index 代表的旋转次数实际上就是 R_SHAPE 数据结构中 shape_r 数组的下标。这个很容易理解，因为我们的穷举算法总是按照一个旋转方向（顺时针方向）遍历 shape_r 数组，所以其下标就代表了旋转次数。

2. 穷举板块的摆放方法

板块摆放方法的穷举分两个步骤，第一个步骤是对板块的每种旋转形态进行遍历，第二步是对每种旋转形态按照从左到右的顺序，依次在每个位置上尝试摆放。第一步比较简单，就是对 R_SHAPE 数据结构中的 shape_r 数组进行遍历。ComputerAIPlayer()函数是 AI 算法的核心，其作用就是模拟人类玩俄罗斯方块游戏的方式将一个指定的板块摆放在最合理的位置上。遍历的第一个步骤，也就是对 shape_r 数组的遍历算法就在这个函数中：

```
bool ComputerAIPlayer(RUSSIA_GAME *game, SHAPE_T s)
{
    bool res_find = false;
    EVA_RESULT best_r = { 0, 0, 0, -999999, -999999 };

    R_SHAPE *rs = &g_shapes[s - 1];
    //遍历每个板块的形状，相当于旋转板块
    for(int i = 0; i < rs->r_count; i++)
    {
        B_SHAPE *bs = &rs->shape_r[i];
        EVA_RESULT evr = { i, 0, 0, -999999, -999999 };
        int rtn = EvaluateShape(game, bs, &evr);
        if((evr.value > best_r.value)
            || ((evr.value == best_r.value) && (evr.prs > best_r.prs)))
        {
            res_find = true;
            best_r = evr;
        }
    }
    if(res_find)
    {
        PutShapeInPlace(game, &rs->shape_r[best_r.r_index], best_r.row, best_r.col);
    }

    return res_find;
}
```

best_r 中存放最终得到的摆放板块的最佳位置和板块的旋转状态，PutShapeInPlace()函数根据这个结果将板块旋转并放置到指定的位置，同时计算消除行并记分。EvaluateShape()函数从游戏区域的 0 列开始，逐个位置尝试摆放这个板块：

```
int EvaluateShape(RUSSIA_GAME *game, B_SHAPE *bs, EVA_RESULT *result)
{
    int start_row = GetTouchStartRow(game, bs);
    if(start_row < 0)
        return -1;

    for(int col = 0; col < (GAME_COL - bs->width + 1); col++)
    {
        int row = start_row;
        //是否还能向下？如果能就再下降一行，直到停下
        while(CanShapeMoveDown(game, bs, row, col))
        {
            row++;
        }
        AddShapeOnGame(game, bs, row, col, true);
        int values = EvaluateFunction(game, bs, row, col);
        int prs = PrioritySelection(game, bs->r_index, row, col);
        RemoveShapeFromGame(game, bs, row, col);
        if((values > result->value)
            || ((values == result->value) && (prs > result->prs)))
        {
            result->row = row;
```

```
                result->col = col;
                result->value = values;
                result->prs = prs;
            }
        }

        return 1;
    }
```

从 0 列开始的遍历在 for 循环内完成，但是在这之前，首先要确定行的起始位置。GetTouchStartRow()函数用于确定起始行位置，计算的依据就是当前的 top_row 和当前板块的高度，从 top_row 指定的位置向上修正板块高度。如果当前 top_row 之上的空间比板块的高度小，则说明没有空间可以摆放这个板块，也就是应该 Game Over 了。CanShapeMoveDown()函数判断板块在这个位置是否还可以继续向下移动，当所在的列存在"井"或开放的"空洞"时，板块是有可能继续向下移动的，所以要处理这种情况。while 循环调用 CanShapeMoveDown()函数，直到不能再下降为止。AddShapeOnGame()函数将板块临时放置在指定为止，然后调用 EvaluateFunction()函数进行评估，完成评估之后，调用 RemoveShapeFromGame()函数取消这次临时放置，使得游戏局面恢复到之前的位置，准备下一个位置的评估。得到一个位置的评估值和优先度值之后，根据评估值的高低更新 result 中的值。

3. 测试我们的 AI 算法

测试的方法非常简单，就是随机生成几千到几十万个板块，然后让我们的 AI 算法一一摆放它们，看看最后能得到什么结果。首先用 GenerateShapeList()函数随机生成 10 万个板块，然后逐个"喂"给代表计算机 AI 的 ComputerAIPlayer()函数。PrintGame()函数打印如图 22-6 所示的一个中间状态。如果将打印输出重定向到一个文件中，可以看到我们的 AI 算法摆放这 10 万个板块的完整过程。

```
std::vector<SHAPE_T> shape_list;
RUSSIA_GAME game;

InitGme(&game);
GenerateShapeList(100000, shape_list);
for(auto i = 0; i < shape_list.size(); i++)
{
    PrintGame(&game, shape_list[i]);
    if(!ComputerAIPlayer(&game, shape_list[i]))
    {
        std::cout << "Failed at: " << i + 1 << " pieces!" << std::endl;
        break;
    }
}
```

我们的 AI 算法最好的结果是消除了 26 万行，平均可以消除 8 万行左右，这比一些优秀的算法差远了。如果你耐心看 ComputerAIPlayer()函数"玩"游戏的整个过程，你会发现这个 AI 经常做出一些"自杀"性举动，说明我们的算法还有很大的改进余地，评估函数还可以继续优化。本章的参考资料里也列举了各种优秀的算法供大家参考。

22

22.4 总结

好了，就是这样，实现一个自动玩俄罗斯方块游戏的 AI 算法非常简单，相信你已经能体会到站在巨人的肩膀上的好处了。不过话又说回来，要做好一个 AI 算法也不是那么容易的。首先我们的算法实现还有很多可以优化的地方，比如数据结构的优化，可以用一维 bit 位组代替二维数组，这样就可以充分利用现代 CPU 的 128 位寄存器和相关的指令优化算法的速度。其次，我们的评估算法还可以再优化，比如使用更好的评估策略，重新定义“空洞”的概念，区分完全封闭的“空洞”和可以填充的开放性“空洞”或者支持板块下落过程中平移或旋转（用于填补侧面的开放性“空洞”）等，这些都是提高算法的 AI 的一些研究方向。

最后，你玩过在线俄罗斯方块对战游戏吗？你被对手虐过吗？你肯定猜他们开挂了，但是是什么原理？你现在知道了吧？

22.5 参考资料

[1] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[2] 维基百科：http://zh.wikipedia.org/wiki/俄罗斯方块

[3] Demaine E D, Hohenberger E, Liben-Nowell D. Tetris is Hard, Even to Approximate. Technical Report MIT-LCS-TR-865, Massachusetts Institute of Technology, 2002.10.21

[4] Bourg D M, Seemann G. *AI Techniques for Game Programming*. Premier Press, 2002

[5] Fahey C. Tetris AI : Computer Plays Tetris. http://colinfahey.com/tetris/tetris.html

[6] Burgiel H. How to lose at tetris. *Mathematical Gazette*, 81:491:194–200, July 1997

[7] Thiery C, Scherrer B. Building Controllers for Tetris. *International Computer Games Association Journal* , March 2010 (2010) 3-11

[8] Bruno Scherrer 和 Christophe Thiery 的算法实现：http://gforge.inria.fr/projects/mdptetris/

第 23 章 博弈树与棋类游戏

1997 年 5 月 11 日，国际象棋世界冠军卡斯帕罗夫在一场挑战赛中以 2.5：3.5 输给了“深蓝”，特别是最后一局，卡斯帕罗夫只走了 19 步就投子认输了。这个结果震惊了全世界，要知道“深蓝”并不是人类，它只是一台几吨重的计算机而已。卡斯帕罗夫之前曾经和“深蓝”的前辈“深思”过招几次，“深思”每次都输得很惨。就在一年前，卡斯帕罗夫还曾经以 4：2 战胜过“深蓝”的一个初级版本。卡斯帕罗夫曾预言计算机在 2010 年之前不可能战胜人类，但是 IBM 的科学家让这个结果提前了 13 年。创新工厂的创始人李开复博士在学校期间，也曾开发过一个黑白棋（Othello）的 AI 算法，据说还战胜了当时美国黑白棋世界冠军。还是那句话：“外行看热闹，内行看门道”，作为程序员我们应该知道这“神奇”的现象的背后一定是某种算法在“作祟”。

棋类游戏通常包含三大要素：棋盘、棋子和游戏规则，其中游戏规则又包括胜负判定规则、落子的规则以及游戏的基本策略。设计一个棋类游戏的 AI 算法，棋盘和棋子的建模是相对比较简单的部分，而游戏规则的建模相对比较复杂。很多情况下，越是简单的规则越难以建模，比如围棋，目前还没有一种有效的理论能够对围棋的“形”和“势”进行建模，使得计算机能像人类一样理解一个围棋棋局。那么棋类游戏的 AI 到底是什么原理？很简单，既然不能让计算机像人一样思考，那就利用计算机强大的计算和数据处理能力搜索结果吧。当然，对于很多棋类游戏来说，穷举搜索所有的棋局是不现实的，比如围棋，因此需要一些理论和算法来支撑搜索工作，这就是本章要介绍的棋类游戏的 AI 算法原理。棋类游戏的人工智能是各种人工智能技术中最基础的一类（或者说根本算不上是 AI），而与博弈树理论相关的各种算法则是棋类游戏人工智能算法的核心，本章将介绍博弈树相关的算法在棋类游戏中的应用。

23.1 棋类游戏的 AI

前面已经提到过，除了棋盘和棋子的建模，棋类游戏最重要的部分就是 AI 算法的设计。目前棋类游戏的 AI 基本上就是带启发的搜索算法，那么，这些搜索算法是建立在什么理论基础上的？常用的搜索算法有哪些？一个棋类游戏的 AI 算法通常都包含哪些内容？本节就来解答这些问题。

23.1.1 博弈与博弈树

首先介绍一下什么是博弈。博弈可以理解为有限参与者进行有限策略选择的竞争性活动，比如下棋、打牌、竞技、战争等。根据参与者种类和策略选择的方式可以将博弈分成很多种，本章讨论的是与棋类游戏有关的简单的“二人零和、全信息、非偶然”博弈，也就是我们常说的零和博弈（Zero-sum Game）。所谓“零和”，就是有赢必有输，不存在双赢的结果。所谓“全信息”，是指参与博弈的双方进行决策时能够了解的信息是公开和透明的，不存在信息不对称的情况。比如棋类游戏的棋盘和棋子状态是公开的，下棋的双方都可以看到当前所有棋子的位置，但是很多牌类游戏则不满足全信息的条件，因为牌类游戏都不会公开自己手中的牌，也看不到对手手中的牌。所谓的“非偶然”，是指参与博弈的双方的决策都是“理智”的行为，不存在失误和碰运气的情况。

在博弈过程中，任何一方都希望自己取得胜利，当某一方当前有多个行动方案可供选择时，他总是挑选对自己最为有利同时对对方最为不利的那个行动方案。当然，博弈的另一方也会从多个行动方案中选择一个对自己最有利的方案进行对抗。参与博弈的双方在对抗或博弈的过程中会遇到各种状态和移动（也可能是棋子落子）的选择，博弈双方交替选择，每一次选择都会产生一个新的棋局状态。假设两个棋手（可能是两个人，也可能是两台计算机）MAX 和 MIN 正在一个棋盘上进行博弈。当 MAX 做选择时，主动权在 MAX 手中，MAX 可以从多个可选决策方案中任选一个行动，一旦 MAX 选定某个行动方案后，主动权就转移到了 MIN 手中。MIN 也会有若干个可选决策方案，MIN 可能会选择任何一个方案行动，因此 MAX 必须对做好应对 MIN 的每一种选择。如果把棋盘抽象为状态，则 MAX 每选择一个决策方案就会触发产生一个新状态，MIN 也同样，最终这些状态就会形成一个状态树，这个附加了 MAX 和 MIN 的决策过程信息的状态树就是博弈树（Game Tree）。博弈树的根就是搜索开始时的棋盘状态，每一个子节点就是 MAX 的每一种决策方案可能产生的棋盘状态（局面），而这些子节点的子节点则是 MIN 的每一种决策方案可能产生的棋盘状态（各层相互间隔）。这棵树的叶子节点就是最终结局，结果无非三种：MAX 胜利、MIN 胜利或者平局。

博弈树的搜索就是从一个棋局状态开始，对每一步棋子移动产生新的棋局状态进行判断，看看是赢还是输，直到最终得到整棵树的判断结果。根据这个搜索过程，如果 MAX 和 MIN 都知道这棵博弈树的全部状态，则结果将变得没有悬念。除非存在一个必胜的（棋局状态）节点序列（就像古老的井字棋游戏那样），否则平局将是所有博弈最后的结局，所有的棋类游戏都将变得无聊至极。幸运的是（反过来也可以理解为不幸），人类的大脑处理不了这么多状态，因此人类对弈的结局依然充满了悬念。对于计算机来说，以目前计算机的处理能力要处理如此多的节点也是不现实的。以中国象棋为例，建立一棵双方各走 50 步的博弈树需要生成大约 10^{160} 个节点，即使处理一个节点只需要 10^{-8} 秒，要处理这棵树也需要 10^{140} 年以上[3]。至于围棋，据估算，其博弈树的节点数大约在 $10^{575} \sim 10^{620}$[3]。由此可见，对于大多数棋类游戏来说，用建立完整的博弈树，从根节点到叶子节点完整地搜索博弈树是不现实的。所以复杂棋类游戏的搜索算法通常都需要指定一个搜索深度，当达到搜索深度时就直接评估棋局，在时间和准确度之间做一个折中。

23.1.2　极大极小值搜索算法

博弈树搜索是各种棋类游戏 AI 算法的基础，极大极小值（Min-Max）搜索算法是各种博弈树搜索算法中最基础的搜索算法。假如 MAX 和 MIN 两个人在下棋，MAX 会对所有自己可能的落子后产生的局面进行评估，选择评估值最大的局面作为自己落子的选择。这时候就该 MIN 落子，MIN 当然也会选择对自己最有利的局面，这就是双方的博弈，即总是选择最小化对手的最大利益（令对手的最大利益最小化）的落子方法。作为一种博弈搜索算法，极大极小值搜索算法的名字就由此而来。

从下棋的角度考虑，是 MAX 和 MIN 双方轮流落子，但是搜索算法的角度考虑，只能以其中一方为基准进行搜索。接下来我们就站在 MAX 的立场上分析一下极大极小值搜索算法的搜索过程。首先我们知道，极大极小值搜索也将得到一棵博弈树，称为极大极小博弈树（Minimax Game Tree）。这棵树的根（第 0 层）是搜索的开始状态。树的第 1 层节点是 MAX 的选择节点，这一层的节点 MAX 将选择对自己最有利的评估最大值，称为极大值节点。树的第 2 层节点是 MIN 选择节点，这一层的节点 MAX 将选择对自己最不利的评估最小值，因为这一层是对 MIN 落子后的局面进行评估，站在 MIN 的立场进行选择，所以这一层的节点又称为极小值节点。极大值节点和极小值节点交错出现在每一层，直到最后一层的叶子节点对棋局进行评估，所谓的叶子节点其实就是搜索达到终局状态或达到指定的搜索深度时的节点。

图 23-1 是简单的井字棋游戏的极大极小博弈树的一部分，第 1 层是极大值节点，三种落子位置得到的评估值分别是–1，0 和–2，MAX 会选择评估值最大的节点，也就是落子在中间位置的局面，这个局面的估值是 1。那么这一层的评估值是怎么得到的呢？那就是根据第 2 层的评估值进行选择。第 2 层是极小值节点，MAX 会选择对自己最不利的局面，也就是说，MAX 对每个分支都会选择评估最小的值作为第 1 层节点的估值。对于第一个分支，MAX 选择–1 作为评估值，对于第二个分支，MAX 选择 1 作为评估值，对于第三个分支，MAX 选择–2 作为评估值，这就是第 1 层三个局面评估值的由来。

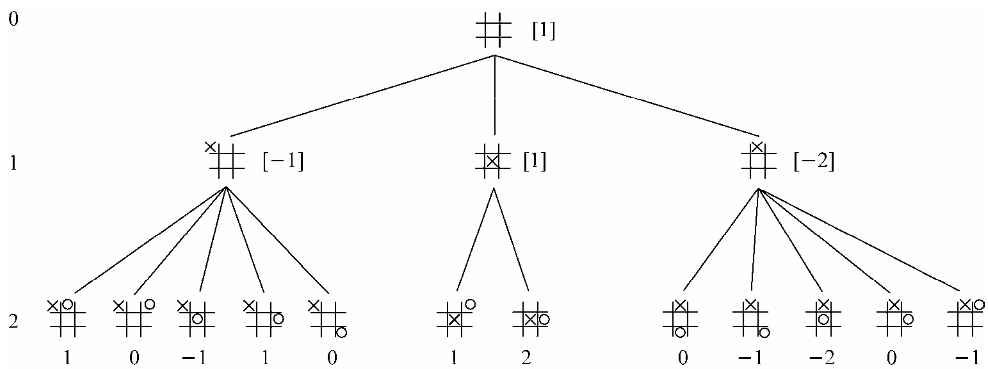


图 23-1　井字棋游戏极大极小博弈树示意图

根据以上分析，我们可以给出极大极小值算法的伪代码：

```
int MiniMax(node, depth, isMaxPlayer)
{
    if(depth == 0)
    {
        return Evaluate(node);
    }

    int score = isMaxPlayer ? -INFINITY : INFINITY;
    for_each(node 的子节点 child_node)
    {
        int value = MiniMax(child_node, depth - 1, !isMaxPlayer);
        if(isMaxPlayer)
            score = max(score, value);
        else
            score = min(score, value);
    }
}
```

有了“α-β”剪枝算法之后，当然不会有人再直接使用极大极小值算法，但它仍然是我们理解其他搜索算法的基础。

23.1.3 负极大极搜索算法

博弈树的搜索是一个递归的过程,极大极小值算法在递归搜索的过程中需要在每一步区分当前评估的是极大值节点还是极小值节点。1975 年 Knuth 和 Moore 提出了一种消除 MAX 节点和 MIN 节点区别的简化的极大极小值算法[5]，称为负极大值算法（Negamax）。该算法的理论基础是：

```
max(a,b) = -min(-a, -b)
```

简单地将递归函数 MiniMax()返回值取负再返回，就可以将所有的 MIN 节点都转化为 MAX 节点，对每个节点的搜索都尝试让节点值最大，这样就将每一步递归搜索过程都统一起来。

根据以上分析，我们可以给出负极大值算法的伪代码，其中 color 参数相当于传递了一个符号位：

```
int NegaMax(node, depth, color)
{
    if(depth == 0)
    {
        return color * Evaluate(node);
    }

    int score = -INFINITY;
    for_each(node 的子节点 child_node)
    {
        int value = -NegaMax(child_node, depth - 1, -color);
        score = max(score, value);
    }
}
```

23.1.4 “α-β”剪枝算法

博弈树搜索算法很简单，但是需要搜索的状态是相当多的。以简单的井字棋（Tic-Tac-Toe）游戏为例，当设定搜索深度是 6 时，不带任何优化的极大极小值搜索算法确定第一个落子时需要搜索 56160 个状态。如果是五子棋或围棋这样的复杂棋类游戏，搜索的状态数会是天文数字，因此需要一些优化方法对简单搜索算法进行优化。“剪枝”是搜索算法中常见的优化方法，通过减除一些明显不可能得到正确解的状态，避免对这些状态的搜索，可以提高搜索算法的效率。本节我们将介绍一种可应用于极大极小值算法和负极大值算法的剪枝算法——“α-β”剪枝算法（Alpha-beta Pruning）。

有很多资料将“α-β”剪枝算法称为“α-β”搜索算法，实际上，它不是一种独立的搜索算法，而是一种嫁接在极大极小值算法和负极大值算法上的一种优化算法。“α-β”剪枝算法维护了一个搜索的极大极小值窗口：[α, β]。其中 α 表示在搜索进行到当前状态时，博弈的 MAX 一方所追寻的最大值中最小的那个值（也就是 MAX 的最坏的情况）。在每一步的搜索中，如果 MAX 所获得的极大值中最小的那个值比 α 大，则更新 α 值（用这个最小值代替 α），也就是提高 α 这个下限。而 β 表示在搜索进行到当前状态时，博弈的 MIN 一方的最小值中最大的那个值（也就是 MIN 的最坏的情况）。在每一步的搜索中，如果 MIN 所获得的极小值中最大的那个值比 β 小，则更新 β 值（用这个最大值代替 β），也就是降低 β 这个上限。当某个节点的 α⩾β 时，说明该节点的所有子节点的评估值既不会对 MAX 更有利，也不会对 MIN 更有利，也就是对 MAX 和 MIN 的选择不会产生任何影响，因此就没有必要再搜索这个节点及其所有子节点了。

“α-β”剪枝算法实际上是两个过程，分别是极小值节点的“α 剪枝”和极大值节点的“β 剪枝”，接下来我们用两幅图分别说明一下这两个剪枝过程的原理。图 23-2 是“α 剪枝”过程示意图。极大值节点 A 搜索博弈树时会从两个极小值节点 B 和 C 中选择评估值最大的一个节点，而 B 和 C 节点则会从自己的子节点中（极大值节点）选择评估值最小的一个节点。假设已经对 B 节点完成了搜索，B 的四个子节点 D、E、F、G 中最小值是 2，则可知 B 节点的准确估值是 2，此时更新 α 的值为 2。接下来开始搜索 C 节点的子节点 H、I 和 J，如果 H 节点的估值是 1，则说明 C 节点的评估值一定不会超过 1（因为 C 总是选择 H、I、J 节点中的最小值），也就是说，C 节点的评估值一定不会比 B 节点的评估值更大，此时就可以终止对 C 节点的搜索，此过程就称为“α 剪枝”。

图 23-3 是“β 剪枝”过程示意图。极小值节点 A 搜索博弈树时会从两个极大值节点 B 和 C 中选择评估值最小的一个节点，而 B 和 C 节点则会从自己的子节点（极小值节点）中选择评估值最大的一个节点。假设已经对 B 节点完成了搜索，B 的四个子节点 D、E、F、G 中最大值是 8，则可知 B 节点的准确估值是 8，此时更新 β 的值为 8。现在开始搜索 C 节点的子节点 H、I 和 J，如果 H 节点的估值是 10，则 C 节点的值一定不会小于 10（因为 C 总是选择 H、I、J 节点中的最大值），也就是说，C 节点的值一定不会比 B 节点更小，因此可以终止 C 节点的搜索，此过程就称为“β 剪枝”。

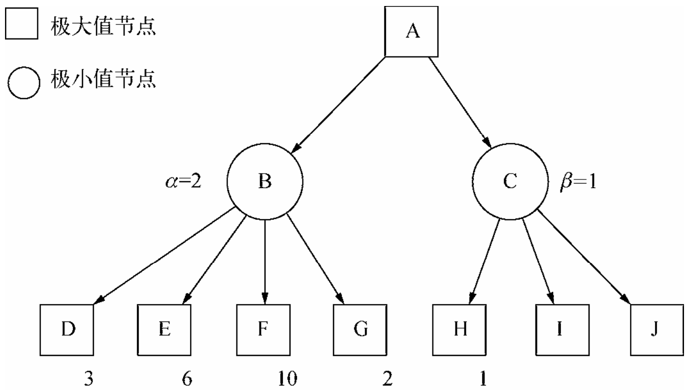


图 23-2　“α 剪枝”过程示意图

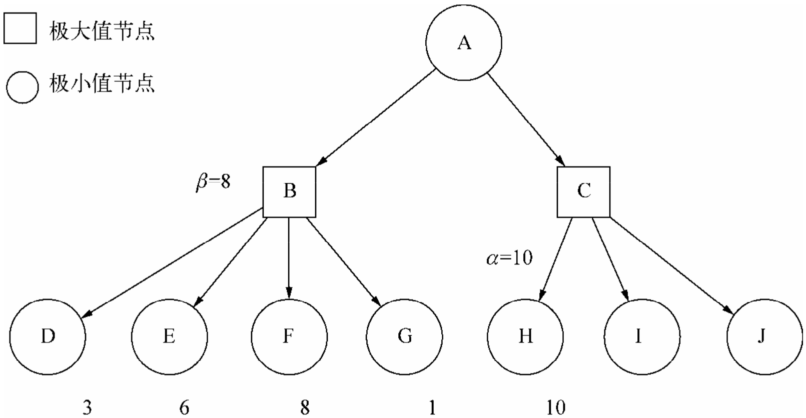


图 23-3　“β 剪枝”过程示意图

这就是“α-β”剪枝算法的原理，搜索开始时，可设定 α=－∞，β=+∞，在搜索过程中，这个范围会逐步收窄，直到出现 α≥β 的剪枝条件。下面我们就给出基于极大极小值算法的“α-β”剪枝算法的伪代码：

```
int MiniMax_AlphaBeta(node, depth, α, β, isMaxPlayer)
{
    if(depth == 0)
    {
        return Evaluate(node);
    }

    if(isMaxPlayer)
    {
        for_each(node 的子节点 child_node)
        {
            int value = MiniMax_AlphaBeta(child_node, depth - 1, α, β, FALSE);
            α= max(α, value);
```

```
            if(α >= β) /*β 剪枝*/
                break;
        }

        return α;
    }
    else
    {
        for_each(node 的子节点 child_node)
        {
            int value = MiniMax_AlphaBeta(child_node, depth - 1, α, β, TRUE);
            β= min(β, value);
            if(α >= β) /*α 剪枝*/
                break;
        }

        return β;
    }
}
```

“α-β”剪枝算法同样可以应用于负极大值算法，需要注意的是，在递归搜索子节点时，需要将极大极小值窗口[α,β]更换为[–β,–α]。应用“α-β”剪枝算法后的负极大值算法伪代码如下所示：

```
int NegaMax_AlphaBeta(node, depth, α, β, color)
{
    if(depth == 0)
    {
        return color * Evaluate(node);
    }

    int score = -INFINITY;
    for_each(node 的子节点 child_node)
    {
        int value = -NegaMax_AlphaBeta(child_node, depth - 1, -β, -α, -color);
        score = max(score, value);
        α= max(α, value);
        if(α>= β)
            break;
    }

    return score;
}
```

23.1.5 估值函数

对于很多启发式搜索算法，其“智力”的高低基本上是由估值函数（评估函数）所决定，棋类游戏的博弈树搜索算法也不例外。博弈树搜索算法基本上就是利用计算机强大的数据处理和计算能力进行蛮力计算，只有在进行棋局评估时才体现出一点点“智力”，这点“智力”就是估值函数的价值。人类的棋手下棋，对棋局都有一个综合评估，协调各个棋子之间的关系，有舍有得，控制棋局向有利的方向发展。但是对于计算机算法来说，把这一套整体的评估和控制转化成一个估值函数，是一个相当复杂的模型。比如弃子是棋类游戏中常用的策略，以退为进，在若干步之

后可获得很好的结果，如何让评估函数也能理解这种策略？

估值函数的作用是把一个棋局量化成一个可直接比较的数字，这个数字在一定程度上能反映取胜的概率。棋局的量化需要考虑很多因素，量化结果是这些因素按照各种权重组合的结果。这些因素通常包括棋子的战力（棋力）、双方棋子占领的空间、落子的机动性、威胁性（能吃掉对方的棋子）、形和势等。不同的棋类游戏会根据规则选择合适的参考因素与权重关系，组合出一个量化的评估结果。权重关系组合在很大程度上决定了估值函数的价值，为了获得一个更好的组合关系，研究者通常会收集成千上万的棋局对自己的估值函数进行“训练”，通过反馈调整权重组合关系。估值函数对棋局量化的原理通常是简单的，但是大多数“战力强悍”的 AI 算法都是不公开自己的估值函数的。

估值函数并不仅仅是简单的评估计算，棋类游戏的估值函数需要综合大量跟棋类有关的知识。相关的知识越少，估值函数越简单，速度快但是效果差。相关的知识越多，估值函数就越复杂，估值函数的质量高但是速度慢。估值算法中增加的知识越多，算法就越慢，很多情况下都需要在速度和质量之间寻求一种平衡。

23.1.6　置换表与哈希函数

置换表（transposition table）也是各种启发式搜索算法中常用的辅助算法，它是一种以空间换时间的策略，使用置换表的目的就是提高搜索效率。结合“α-β”剪枝算法，直接通过置换表可以获得该节点的一个已经缩小范围的搜索窗口，直接在这个搜索窗口上进行搜索可以提高剪枝的效率。如果通过置换表可以得到该节点的一个明确的搜索结果（通常这个结果是当前已知的最好结果），则可直接利用这个结果，没有必要再对这个节点进行搜索。本节我们将介绍与置换表相关的一些知识。

1. 置换表的原理

一般情况下，置换表中的每一项代表者一个棋局中最好的落子方法，直接查找置换表获得这个落子方法能避免耗时的重复搜索，这就是使用置换表能大幅提高搜索效率的原理。

置换表用于存储已经搜索过的棋局（包括以该棋局为根的搜索子树）的搜索结果。搜索算法在搜索一个棋局时，首先查置换表，如果从置换表中能查到这个棋局的信息（已经完成的搜索结果），就可以直接使用这些信息，从而避免对这个棋局再次做完整搜索。置换表的每个表项包含与该棋局有关的搜索信息，这些信息包括评估结果、搜索深度、落子方法和位置等信息。

如果该棋局及其状态子树已经完全搜索，则会存储该棋局的精确结果，如果该棋局及其状态子树还没有完成搜索，则会存储已经完成的搜索窗口。使用“α-β”剪枝的搜索算法通常有三种不同类型的评估值，分别是精确值、α 值和 β 值。精确值，顾名思义，就是搜索得到的评估结果落在区间[α, β]之内，就将评估结果视为精确值。如果状态子树的所有子节点没有找到比当前极大值更好的结果，则将评估结果视为 α 值。如果状态子树的所有子节点没有找到比当前极小值更差的结果，则将评估结果视为 β 值。

搜索深度也是置换表中的一个重要属性，它决定了对这个表项的使用和更新方式。假如要对一个棋局进行 n 层深度的搜索，如果置换表中存在一个搜索深度是 m，且 $m \geqslant n$ 的表项，则说明这个棋局的搜索结果可以直接使用，无需对该棋局再做完整的搜索。

除此之外，落子方法和位置用于指导落子和修改棋局状态，也是很重要的信息。

2. 哈希算法

使用置换表最大的问题是置换表的组织和查找的效率。一般来说，置换表越大，查找的命中率就越高。但这个关系不是绝对的，当置换表大小达到一定规模后，不仅不会再提高命中率，反而会因为耗时的查找操作影响算法的效率。所以置换表不是越大越好，需要根据计算机的性能以及搜索的深度选择一个合适的大小。此外，为了查找操作更高效，通常都会用可直接访问的哈希表方式组织置换表，哈希函数的性能就成为影响置换表性能的重要因素。

棋类游戏普遍采用 Zobrist 哈希算法，在本书的第 20 章已经介绍过 Zobrist 哈希算法的原理和实现，本章在介绍黑白棋和五子棋的搜索算法时，会再次用到 Zobrist 哈希算法。

3. 置换表的替换原则

置换表的替换原则，也称覆盖策略，就是同一个棋局（棋局的哈希值相同）如果有了更新的搜索结果，以何种方式更新置换表中的表项。对于单一的置换表算法，其替换原则一般有两种，一种是深度优先替换（deeper priority），一种是始终（随时）替换（always replace）。深度优先替换原则执行的是“同样的搜索深度或更深时替换”的策略，也就是说，只有新棋局的搜索深度大于或等于置换表中已经存在的值时，才更新置换表中的值。深度优先策略只考虑搜索的深度，没有考虑棋局演化出的新棋局信息对后续演化的影响，置换表容易被已经过时但是搜索深度很深的棋局占满，无法保证棋局评估结果的实时性，同时也降低了置换表的搜索效率。始终替换原则就是不考虑其他情况，如果置换表中存在搜索过的棋局，始终用新的搜索结果替换已经存在的结果。始终替换策略总是用新的结果代替旧的结果，能保证棋局评估结果的实时性，但是容易丢掉搜索层数较深的棋局评估结果，而搜索深度越深，往往意味着更优的评估值（对很多搜索算法而言，结果往往是这样的）。

两种原则各有优缺点，有没有一种能将二者结合在一起的策略呢？很多研究者[14][15]在这方面做了很多深入的研究，不过最简单、也是最常用的策略就是使用双置换表。简单来说，双置换表就是使用两个单置换表，一个使用深度优先策略，一个使用始终替换策略。查表时每次查找两个表，只要一个表中查到结果就可以直接使用，如果两个表中都查到结果，就根据实现制定的顺序策略选择其中一个。更新的时候，应用两种策略分别对两个置换表同步更新。

当然，也有一些开源的棋类软件使用了分类置换表算法，就是同一个棋局的置换表对应多个值，按照搜索深度从大到小排序，每当更新一个棋局时，将新的搜索结果按顺序插入到对应的位置中，同时删除搜索深度最小的那个结果。如果新结果的搜索深度小于当前最小的搜索深度，则直接替换当前搜索深度最小的结果。原理上有点像对双置换表的扩展，将其扩展成 n 层置换表，

但是搜索和使用置换表的方法仍然是相同的。

23.1.7 开局库与终局库

俗话说："好的开始是成功的一半。"棋类游戏的开局尤其如此。如果能在开局阶段占据先机，对整个棋局的发展都是非常有利的。终局又称残局，是棋类游戏中决定胜负的最后阶段，也是棋类游戏中非常重要的一个阶段。在开局和终局阶段，棋盘上的变化与正常进行的中局有显著的不同，比如棋子的数量、某些类型的棋子的走法（比如中国象棋的"兵"和"卒"）都会发生变化。很多棋类游戏在开局和终局阶段棋盘上的棋子很少，棋类游戏 AI 的搜索算法在面对空荡荡的棋盘时，常用的启发手段基本上失效，搜索算法退化为普通的穷举搜索，很多落子位置最终的评估结果都是一样的，搜索算法变得"不知所措"，也正是这个原因导致许多棋类游戏的 AI 算法在开局阶段或终局阶段常常走出令人匪夷所思的"昏招"。针对这种情况，很多棋类游戏的算法都会使用开局库和终局库，在开局和终局阶段，直接从库中搜索已知的开局和残局走法，借鉴各种经典的和成熟的开局走法，利用前人对弈的智慧度过这个阶段。到进入中局时，棋盘上的棋子比较多，在搜索过程中可以利用各种启发式搜索获取千变万化的棋局的评估结果时再使用搜索算法。

所谓的开局库和终局库实际上就是一种存储了各种开局和终局棋局信息的数据库。以开局库为例，库中存储了很多已知的经典开局，都是一些很有规律的定势。棋类游戏的 AI 在对弈的开始阶段都从开局库中搜索落子方法，直到棋局演化的局面无法在开局库中找到对应的落子方法为止，此时算法才开始真正的搜索。开局库一般都存些什么内容呢？开局库一般要存储开局的棋局，该棋局对应的各种走法和评估分数，有些开局库还统计了该开局最终的胜局次数、平局次数和负局次数，给出开局棋局的权重等附加信息供搜索时选择。

终局决定了一盘棋的胜负，终局中也有很多规律和定势，许多棋类游戏算法也会使用终局库，以便在终局阶段借鉴一些经典的走法。相对于局面简单的开局库，终局库棋子没有固定的位置，走法更为多样化，棋局的变化更无常，因此终局库的规模常常是开局库的几百或几万倍，检索时间比较长，效率比较低，需要根据实际需要酌情使用。

23.2 井字棋——最简单的博弈游戏

井字棋游戏在西方又被称为 Tic-Tac-Toe，是一种简单的九宫格游戏，因其棋盘很像汉字的"井"字而得名。井字棋游戏玩法是在 3×3 的 9 个方格子棋盘上，两人持不同颜色的棋子交替落子，谁的棋子在横、竖和交叉方向先连成 3 个就算获胜。在 2.1 节我们介绍了棋类游戏的 AI 算法相关的一些理论和算法设计，本节我们就结合这个井字棋游戏设计一个简单的人机博弈游戏。虽然简单，但是包含了一个棋类游戏需要解决的基本问题，比如棋盘和棋子状态建模、博弈树搜索算法设计、静态棋局评估函数和如何产生井字棋的走法（落子方法）等。

通过一个简单的估值函数加上博弈树搜索就使计算机具备了与人玩井字棋游戏的能力，虽然智力不高，但是计算机确实是在玩井字棋游戏，来看看怎么做吧。

23.2.1　棋盘与棋子的数学模型

井字棋的棋盘是 3×3 的九宫格，比较容易想到用一个 3×3 的二维数组表示棋盘，而数组的值就是棋盘上棋子的状态。使用二维数组的好处是数据访问比较直观，二维数组的两个下标可以直接表示棋子的位置。但是使用二维数组的缺点也是明显的，首先是遍历棋盘需要用两重循环处理两个下标，其次是判断行、列以及斜线方向上是否满足三子一线的算法不统一，根据行、列和斜线的下标变化特点，需要用几套不同的方法处理。现在换个思路，用长度为 9 的一维数组表示 3×3 的棋盘如何？这样做损失的是数据访问的直观性，比如第二行第一个棋盘格，对应的数据存在数组的第四个元素中。但是使用一维数组的好处是处理数据简洁，只需对数组一维遍历就可以得到棋盘的当前状态，不需要关注两个下标的计算，最重要的是，结合一点小技巧可以用一套统一的算法非常简洁地处理上面提到的判断三子一线的问题。

有了棋盘和棋子的状态，加上当前落子的玩家 ID，即可构成一个棋局在某一时刻的状态。以下就是棋局状态的定义：

```
class GameState
{
    …
    Evaluator *m_evaluator;
    int m_playerId;
    int m_board[BOARD_CELLS];
};
```

m_evaluator 是评估算子，是对棋局估值的委托算子，它不属于棋局状态，但是从代码实现角度理解，可以作为棋局对象的一个属性。m_board[i]的值有两种状态，即空的状态和有棋子的状态。空状态时其值是 PLAYER_NULL，有棋子的状态时其值是玩家的 ID，所以 m_board[i]的值可能为 PLAYER_NULL、PLAYER_A 或 PLAYER_B 三种情况。

以上就是棋局状态的数据结构定义，现在来看看前文提到的判断三子一线的小技巧。观察井字棋游戏的棋盘，能够排成三子一线的情况一共有三横、三竖加两条斜交叉线 8 种情况，我们事先把这 8 种情况的数组下标组织成一个表：

```
int line_idx_tbl[LINE_DIRECTION][LINE_CELLS] =
{
    {0, 1, 2}, //第一行
    {3, 4, 5}, //第二行
    {6, 7, 8}, //第三行
    {0, 3, 6}, //第一列
    {1, 4, 7}, //第二列
    {2, 5, 8}, //第三列
    {0, 4, 8}, //正交叉线
    {2, 4, 6}, //反交叉线
};
```


这样在判断三子一线时，不需要做复杂的数组下标计算，直接查这张表就可以依次判断 8 条线上是否有三子一线的情况。GameState 类中判断三子一线的算法就是这么实现的：

```
bool GameState::CountThreeLine(int player_id)
{
```

```
    for(int i = 0; i < LINE_DIRECTION; i++)
    {
        if( (m_board[line_idx_tbl[i][0]] == player_id)
            && (m_board[line_idx_tbl[i][1]] == player_id)
            && (m_board[line_idx_tbl[i][2]] == player_id) )
        {
            return true;
        }
    }

    return false;
}
```

这就是算法设计中常用的用数据表进行一致性处理的技巧，在本书的其他章节中也多次用到这种技巧。使用精心构造的数据表，可以让很多棘手的问题的实现代码变得无比简单。

23.2.2 估值函数与估值算法

研究井字棋游戏的估值函数，需要理解井字棋游戏的一些棋局现象。首先是空行数的概念，所谓棋子占据的空行数，指的是棋子所在的行、列或斜线方向上只有己方的棋子或空格子的行（列、斜线）数 + 全是空格的行（列、斜线）数。如图 23-4 所示，第一个棋局中 X 棋子的空行数是 5，O 棋子的空行数是 4，第二个棋局中 X 棋子的空行数是 4，O 棋子的空行数是 3。根据对井字棋游戏规则的理解，对于一个井字棋的棋局，每个玩家的棋子占据的空行越多，就说明该玩家有更大的可能性凑成三子一线的结果，因此空行数是井字棋游戏估值函数评估棋局的一个重要因素。

但是井字棋游戏的评估并不是只考虑空行数这一个因素，在某些情况下，一方棋子占据的空行数多并不一定说明局面占优势，因为井字棋游戏还存在了双连子的情况。所谓的双连子，指的是在一行（列、斜线）上有两个己方的棋子而没有对方的棋子的情况。如图 23-4 的第二个棋局，O 的棋子形成了双连子而 X 的棋子不是双连子。在这种情况下，尽管 X 棋子的空行数比 O 棋子的空行数多，但是 O 棋子形成了双连子，比 X 棋子更有优势。

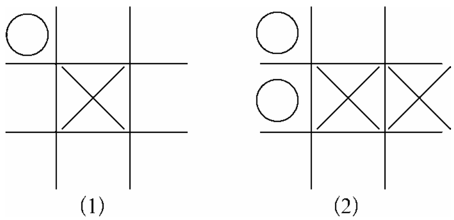


图 23-4 井字棋局面示意图

综上考虑，我们给出了一种井字棋游戏的估值函数计算方法，对于执 X 棋子的一方来说，其评估函数是：

$$E(\mathrm{X})=\begin{cases}+\infty \\ -\infty \\ 0 \\ (\mathrm{X}双连子数-\mathrm{O}双连子数)\times 10+(\mathrm{X}空行数-\mathrm{O}空行数)\end{cases}$$

+∞表示 X 获胜的局面，−∞表示 X 失败的局面，0 表示双方是平局，其他值是具体的评估值。根据这个估值函数，我们设计了 FeEvaluator 算子，以下是 FeEvaluator 算子估值函数的算法实现：

```
int FeEvaluator::Evaluate(GameState& state, int max_player_id)
{
    int min =  GetPeerPlayer(max_player_id);

    int aOne, aTwo, aThree, bOne, bTwo, bThree;
    CountPlayerChess(state, max_player_id, aOne, aTwo, aThree);
    CountPlayerChess(state, min, bOne, bTwo, bThree);

    if(aThree > 0)
    {
        return INFINITY;
    }
    if(bThree > 0)
    {
        return -INFINITY;
    }

    return (aTwo - bTwo) * DOUBLE_WEIGHT + (aOne - bOne);
}
```

井字棋游戏的估值算法很多，有很多网友也提供了其他方法，在本章的随书代码中，还用另一种方式实现了一个 WzEvaluator 算子，经过测试，两个评估算子的棋力差不多。

23.2.3 如何产生走法（落子方法）

搜索算法负责对棋局进行评估，选择下一步的最佳落子位置，但是搜索过程中需要遍历所有可能的棋子落子或移动方法，这就需要一种能够推动落子或棋子移动的算法。棋类游戏的规则多种多样，有的只能放置棋子，不能移动棋子，有的只能移动已有的棋子，因此走法产生算法也是多种多样的，很难找到通用的算法。井字棋游戏的走法产生非常简单，就是对 9 个空格中还没有放置棋子的格子依次进行落子试探即可，所有空格子都试过以后，走法产生的算法就结束了，非常简单。

走法产生算法一般配合搜索算法，成为搜索算法的一部分。下面我们就给出井字棋游戏的极大极小值搜索算法实现，内含了走法产生，结合 23.1.2 的算法解释，这段代码不难理解。MiniMax() 函数就是极大极小值搜索算法的实现，其中的 for 循环就是走法产生算法。为了配合 23.1 节的内容，井字棋游戏还实现了带“α-β”剪枝的搜索算法 AlphaBetaSearcher 和负极大值搜索算法

NegamaxSearcher，有兴趣的读者可以查看本章的随书代码，了解这两种搜索算法的实现细节。通过对比，“α-β”剪枝算法对提高搜索效率方面确实有非常好的效果。对于空盘状态的棋局，假如设置搜索深度是 6，极大极小值搜索算法搜索棋局的次数是 65000 多次，但是应用“α-β”剪枝后搜索棋局的次数只有 6500 多次。

```
int MinimaxSearcher::MiniMax(GameState& state, int depth, int max_player_id)
{
    if(state.IsGameOver() || (depth == 0))
    {
        return state.Evaluate(max_player_id);
    }

    int score = (state.GetCurrentPlayer() == max_player_id) ? -INFINITY : INFINITY;
    for(int i = 0; i < BOARD_CELLS; i++)
    {
        GameState tryState = state; /*生成临时棋局状态对象*/
        if(tryState.IsEmptyCell(i))/*此位置可以落子*/
        {
            tryState.SetGameCell(i, tryState.GetCurrentPlayer());
            tryState.SwitchPlayer();
            int value = MiniMax(tryState, depth - 1, max_player_id);
            if(state.GetCurrentPlayer() == max_player_id)
            {
                score = std::max(score, value);
            }
            else
            {
                score = std::min(score, value);
            }
        }
    }

    return score;
}
```

实现了搜索算法和评估算子，结合专为本书而做的一个棋类游戏代码框架（参见附录 B），就可以实现一个简单的人机对战井字棋游戏（如图 23-5 所示）。来看看结果吧，设定搜索深度为 6 的时候，计算机的智商貌似不错，我最多只能玩个平局。

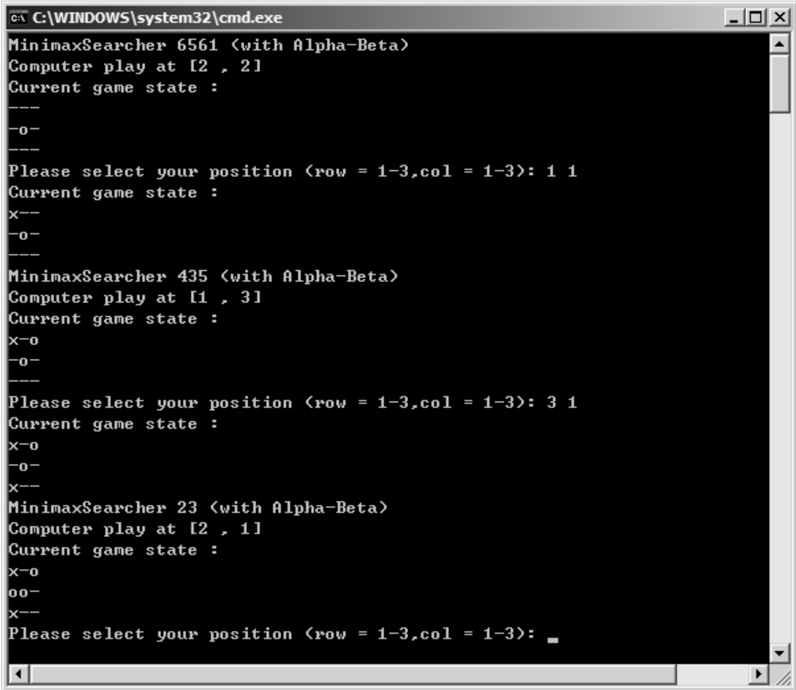


图 23-5　简单的井字棋游戏

23.3　奥赛罗棋（黑白棋）

奥赛罗棋（Othello）又称黑白棋、翻转棋（Reversi），在西方和日本非常流行。游戏双方分别执黑白两种颜色的棋子，在 8 × 8 的棋盘上轮流落子，相互翻转对方的棋子。只要落子和棋盘上任一枚己方的棋子在横、竖和斜方向上能夹住对方棋子，就能将对方的这些棋子转变为己方棋子。如果一方在任一位置落子都不能夹住对手棋子，就要让对手下子，如果双方皆不能落子，则游戏结束，棋盘上棋子多的一方取胜。奥赛罗棋游戏规则简单，很容易上手，但是要玩得好就不容易了。

为了便于识别棋子的位置，用数字 1 ~ 8 标识棋盘的行，用字母 A ~ H 标识棋盘的列，如图 23-6 所示。黑白棋游戏的规则比较特殊，有时候一个落子就会造成十几个子的翻转，因此很容易出现双方比分剧烈变化的情况。即使在游戏的前期不占优势，只要占据了有利位置，后期很可能几个回合就能将对方大量的棋子翻转为己方棋子，从而扭转局势。因此黑白棋游戏的前期一般不太着眼于子的多少，更重要的是棋子的位置。中间位置的棋子最容易受到夹击，在横、竖和斜线共四个方向上都可能被夹击。边缘的棋子则只有一个可能被夹击的方向（横向或竖向），而四个角上的棋子则完全不可能被夹击，是最安全的位置。正因为这样，黑白棋有“金角银边草肚皮”之说。

C 位（C-squares）、星位（X-squares）、角和边是黑白棋中的一些特殊的位置。在图 23-6 中标记了字母 C 的 A2、A7、B1、B8、G1、G8、H2 和 H7 几个位置即为 C 位，标记了字母 X 的 B2、B7、G2 和 G7 四个位置即为星位。下棋过程中不到万不得已不要占用 C 位和星位，因为对手可能会借助 C 位和星位的己方棋子做桥梁占领相邻的角位置。与 C 位相比，星位的危害更大，因为星位上如果落了己方的棋子，对手就可以从 5 个方向攻击相邻的 C 位和角位置。标记了字母 E 的是边位置，这些位置相对不容易受到攻击，下棋过程中应考虑优先落子在这些位置。与之对比的与边相邻的标记了字母 S 的位置，这些位置容易导致对手占领边位置，因此下棋时尽量不要落在这些位置。

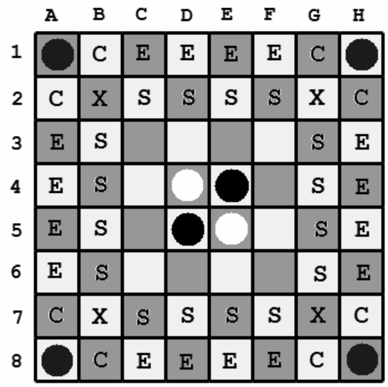


图 23-6　黑白棋棋盘位置示意图

除了这四个位置，黑白棋中还有几个重要的概念，比如内部子（internal discs）、边缘子（external discs）、稳定子（stable discs）和行动力（mobility）等。不与空位相邻的棋子被称为内部子，与之相对的就是边缘子。当对手落子时，边缘子就是直接被夹击的对象，内部子相对好一些，边缘子和内部子都是考察一个黑白棋局面的重要参考要素，没有内部子的局面通常是个糟糕的局面，边缘子太多同样糟糕。在棋盘上绝对不会被翻转的棋子就是稳定子，稳定子越多，局面越有利，四个角位置上的棋子就是天然的稳定子。一般来说，黑白棋的前 20 手一般不会出现稳定子，所以有一些黑白棋估值理论通常在黑白棋开局的时候不考虑稳定子的因素。最后是行动力的概念，行动力是指合法的落子位置的数量，当一方拥更多的合法落子位置可供选择时，就意味着其具有更好的行动力。

23.3.1　棋盘与棋子的数学模型

黑白棋的棋盘是 8 × 8 个格子，很容易联想到用二维数组来表示棋盘和棋子状态。23.2.1 节介绍井字棋游戏时，已经提到过使用二维数组虽然展示更直观一些，但是在横向、竖向和斜线方向搜索棋子状态时会遇到算法不一致问题的困扰，因此对于黑白棋的棋盘和棋子状态，我们仍然使用一维数组来建模。黑白棋游戏比井字棋游戏复杂很多，有更多的规则需要判断，使用一维数组为棋盘和棋子状态建模，需要很多技巧。本节我们不另辟蹊径，直接借用 Warren Smith 提出的一种建模方法[20]，接下来我们就详细介绍一下 Warren Smith 提出的棋盘与棋子模型。

1. Warren Smith 棋盘状态模型

Warren Smith 的模型使用一个长度为 91 的一维数组表示黑白棋的棋盘与棋子状态，其中 64 个是棋盘上的位置，27 个是标志位或哨兵位。91 个数组元素中前 10 个和后 10 个是标志位，中间每间隔 8 个位置插入一个标志位，这个模型各个位置的逻辑结构如下阵列所示：

```
ddddddddd
dxxxxxxxx 10
```

```
dxxxxxxxx 19
dxxxxxxxx 28
dxxxxxxxx 37
dxxxxxxxx 46
dxxxxxxxx 55
dxxxxxxxx 64
dxxxxxxxx 73
dddddddddd
```

字母 d 标识的就是标志位，用特殊值 DUMMY 表示，*x* 是棋子状态，我们的算法用 PLAYER_A 和 PLAYER_B 分别表示双方的棋子，用 PLAYER_NULL 表示空位置。一般人会觉得应该用标志 d 把整个棋盘都框起来，这相当于在每 8 个棋盘位置中间插入两个标志位，其实没有这个必要，一个标志位就足以保证在任意一个 *x* 位置向 8 个方向搜索都能遇到标志位而正常结束。如果你还有疑问，看完下面对方向步进数组的介绍后就能明白这样设置标志位的原因了。

井字棋游戏比较简单，我们介绍的算法用一个表预置了 8 个行、列和斜线的方向，但是这个方法不适用于黑白棋，因为黑白棋的行、列和斜线的组合太多了。尽管如此，我们还是有办法避免像二维数组那样需要分别用行和列的下标步进来处理各种方向，其窍门就是使用方向步进数组。对于任意一个 *x* 位置，向右搜索意味着每次 *x* 的下标+1，向左搜索意味着每次 *x* 的下标–1，向上搜索意味着每次 *x* 的下标–9，向下搜索意味着每次 *x* 的下标+9。斜线方向也是一样，最终的方向步进数组可以定义为：

```
const int dir_inc[] = {1, -1, 8, -8, 9, -9, 10, -10};
```

现在明白了吧，对于上述阵列，用这个不仅数组向任意一个方向步进，最终都会遇到标志位而自然结束，这正是这个模型的高明之处。

现在问题是，这个模型中的元素与实际棋盘上的行和列的坐标如何换算呢？其实很简单，就是：

```
square(row,col) = board[10+col+row×9]    （0<= row,col <=7）
```

有了这个关系，就不难将棋盘上的行、列坐标与棋盘模型中的一维数组元素对应起来了。

除了棋盘和棋子的状态，黑白棋的棋盘上空位是一个比较重要的数据，黑白棋的落子都是在空位上进行的。虽然可以搜索棋盘得到每个空位的位置，但这不是一种高效的做法。通常的做法是用一个列表将这些空位组织起来，在搜索算法中直接使用这个空位表进行搜索，要比搜索整个棋盘得到这些空位信息要快很多，特别是进入中局阶段以后，空位的个数逐渐减少，这个表越来越小，这种组织方法所带来的效率提升作用就更明显。很显然，在搜索过程中这个表将面临频繁的插入和删除操作，因此我们使用双向链表来组织这个空位列表，这个链表定义如下：

```
typedef struct tagEMPTY_LIST
{
    int cell;
    tagEMPTY_LIST *pred;
    tagEMPTY_LIST *succ;
}EMPTY_LIST;
```

其中 cell 就是这个空位在一维棋盘模型数组中的位置。

2. Warren Smith 模型示例

这一节，我们用一个判断一个落子是否合法的算法来介绍一下如何使用这个棋盘模型，通过对比可以清楚地理解这个模型的优点。落子是否合法就是看是否能翻转对手的棋子，只要在 8 个方向上的任意一个方向上能翻转对手的棋子即为合法落子位置。判断一个方向是否能翻转对手棋子的方法是：首先这个落子位置应该是个空位，如果在这个方向上与此空位相邻的是对方的棋子则沿着这个方向继续搜索，直到遇到的棋子不是对方的棋子为止。如果此时遇到的棋子刚好是己方的棋子，则说明在这个搜索方向上可以翻转对方的棋子。当然，遇到的这个棋子也可能是另一个空位置或哨兵位，这就说明在这个搜索方向上不能翻转对方的棋子。下面我们就给出沿一个方向搜索是否能翻转对手棋子的算法：

```
bool GameState::CanSingleDirFlips(int cell, int dir, int player_id, int opp_player_id)
{
    int pt = cell + dir;
    if(m_board[pt] == opp_player_id)
    {
        while(m_board[pt] == opp_player_id)
        {
            pt += dir;
        }

        return (m_board[pt] == player_id) ? true : false;
    }

    return false;
}
```

在这个函数参数中，cell 是搜索开始的空位置在一维数组中的位置，dir 是方向步进值，就是 dir_inc 数组中的某个值，player_id 是当前落子的玩家 ID（也就是棋子的值），opp_player_id 是对手的棋子的值。算法原理非常简单，m_board 的下标 pt 沿着 dir 方向步进，直到下一个棋子不是 opp_player_id 时终止步进，并判断这个棋子是否是 player_id，如果是则返回 true，否则返回 false。

沿着 8 个方向分别搜索，只需要用 CanSingleDirFlips()函数依次遍历 dir_inc 数组即可。当然，并不是所有的位置都需要遍历 8 个方向，比如四个角上的位置，就只需要遍历 3 个方向，而边上的位置则只需要遍历 4 个方向。为了提高算法效率，Warren Smith 的模型中还引入了一个方向掩码表，对棋盘模型中的 91 个位置都定义与之对应的方向掩码。方向掩码用一个字节表示，这个字节中的每个比特对应 dir_inc 数组中的一个方向。如果这个比特是 1，则表示相对于这个位置来说这个方向是有效方向，需要搜索。如果这个比特是 0，则表示这个方向不需要搜索。最终定义的方向掩码表如下：

```
unsigned char dir_mask[BOARD_CELLS] =
{
0,0,0,0,0,0,0,0,0,
```

```
0,81,81,87,87,87,87,22,22,
0,81,81,87,87,87,87,22,22,
0,121,121,255,255,255,255,182,182,
0,121,121,255,255,255,255,182,182,
0,121,121,255,255,255,255,182,182,
0,121,121,255,255,255,255,182,182,
0,41,41,171,171,171,171,162,162,
0,41,41,171,171,171,171,162,162,
0,0,0,0,0,0,0,0,0
};
```

有了这个表，对一个空位完成 8 个方向搜索的算法实现就非常简洁高效了，可以用一个掩码过滤掉一些方向。请看 CanFlips()函数的实现：

```
bool GameState::CanFlips(int cell, int player_id, int opp_player_id)
{
    /*在 8 个方向试探，任何一个方向可以翻转对方的棋子就返回 true*/
    for(int i = 0; i < 8; i++)
    {
        unsigned char mask = 0x01 << i;
        if(dir_mask[cell] & mask)
        {
            if(CanSingleDirFlips(cell, dir_inc[i], player_id, opp_player_id))
            {
                return true;
            }
        }
    }

    return false;
}
```

23.3.2 估值函数与估值算法

黑白棋游戏有很多可用于估值的参数，除了前面介绍的边和角的位置关系、边缘子与内部子、稳定子以及行动力等几个概念，还有前沿子（潜在行动力）、棋子数以及奇偶性等因素。这么多参考因素是否都需要参与评估？如何参与评估？以及它们在最后的估值中占的比重就是估值函数设计的重点。本节我们将介绍黑白棋游戏 AI 中常用的估值函数模型，以及我们最后所用的估值函数算法设计。

1. 常见的估值模型

黑白棋有很多估值函数模型，Gunnar Andersson 在他的文章“Writing an Othello program”中提到了三种常用的估值函数模型，分别是基于位置价值表的估值模型（Disk-square tables）、基于行动力的估值模型（Mobility-based evaluation）和基于模板的估值模型（Pattern-based evaluation）。首先介绍一下基于位置价值表的模型，这个模型的着眼点在于棋盘上每个位置都有不同的价值，四个角的价值最高，与角相邻的几个位置价值最低，以此类推，给棋盘上的每个位置都定一个价值分。有些位置甚至给出一个负价值分表示惩罚性记分，比如星位。评估时根据每个棋子所在的

位置的价值分求和，给出评估结果。有一些复杂的模型甚至在棋局的不同阶段使用不同的位置价值表，比如角位置，其在开局和中局阶段的重要性要比终局阶段要高。单纯使用位置价值表忽略了太多黑白棋游戏的评估因素，算法的 AI 一般不高，当然，这个估值模型的优点是简单。

大多数人类黑白棋棋手下棋时最关注的就是己方的行动力和前沿子数量，棋手们总是追求最大行动力和最少前沿子数量，这就是基于行动力的估值模型的理论基础。当然，有一些基于行动力的评估模型会同时考虑边和角的关系，并在游戏的早期阶段使用一些策略避免己方的棋子过多，这是对基于行动力的评估模型的扩展。

行动力，边沿子（潜在行动力）和稳定子是黑白棋估值的三个重点参数，但是这些参数的计算都比较复杂，精确地计算这些参数往往影响评估的速度。为了加速估值算法，人们提出了基于模板的估值模型。模板估值模型的思想是将全局的行动力，边沿子和稳定子化为局部的行动力，边沿子和稳定子，再将这些局部的参数组合来表示全局参数。每个局部包含的棋子个数不多，可以预先计算好，这样在最终估值时就可以用查表代替计算，如此来加快速度。以 Zebra 程序为例，它将一个 8×8 的棋盘，剪切成 13 种不同的模板，每种模板都有不同的实例，一共有 46 种不同的模板实例。每一个棋局，都可以由这 13 种模板得到的 46 个不同的模板实例对棋局进行估值并相加而得到总的估值。那么这 46 个模板实例（系数值）是如何得到的呢？答案就是用大量的棋局进行训练。具体如何定义和训练模板，请参考 Michael Buro 的文章“Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello”[25]，此处不再赘述。训练充分的好模板给出的估值都比较精确，估值效果与直接计算这三个参数不相上下，但是由于速度快，可以进行更大深度的搜索，通常具有更好的棋力。

2. 估值函数实现

现在该讨论我们的估值函数了。本章的例子不追求多强的棋力，只关注算法的实现，因此我们采用一个简单的评估策略。我们的算法根据棋盘上空位的数量，将棋局粗略地分为开局、中局和终局三个阶段。当棋盘上的空位大于 40 个时，被认为是开局阶段，因为此阶段棋盘上的棋子比较少，可参考的位置因素影响不大，此阶段的评估只考虑行动力因素。当棋盘上的空位大于 18 且小于 40 时，被认为是终局阶段，这个阶段开始考虑棋子在棋盘上的位置估值，同时结合行动力进行评估，二者的评估系数分别是 2 和 7。当棋盘上的空位小于 18 个时，被认为是终局阶段，此时除了考虑位置估值和行动力之外，还考虑对棋子数量进行评估，但是会给棋子数量一个比较低的评估系数。

行动力可以理解为一方可落子的位置数量，计算行动力就是遍历所有空位置，考察每个空位置是否是合法的落子位置（能翻转对手的棋子）。23.3.1 节已经给出了判断一个空位是否能落子的算法，计算行动力的算法可以简单实现如下：

```
int GameState::CountMobility(int player_id)
{
    int opp_player_id = GetPeerPlayer(player_id);
    int mobility = 0;
```

```
    for (EMPTY_LIST *em = m_EmHead.succ; em != NULL; em = em->succ )
    {
        if(CanFlips(em->cell, player_id, opp_player_id))
        {
            mobility++;
        }
    }

    return mobility;
}
```

m_EmHead 是空位链表的头节点，通过 m_EmHead 遍历空位列表，调用 CanFlips()函数判断每个空位能否落子。

进入中局时需要计算棋子位置的价值，我们根据国外的资料整理了一个棋子位置的价值表，如下所示：

```
int posValue[BOARD_CELLS] =
{
    0,0,0,0,0,0,0,0,0,
    0,100,  -8,  10,   5,   5,  10,  -8, 100,
    0,-8, -45,   1,   1,   1,   1, -45,  -8,
    0,10,   1,   3,   2,   2,   3,   1,  10,
    0,5,   1,   2,   1,   1,   2,   1,   5,
    0,5,   1,   2,   1,   1,   2,   1,   5,
    0,10,   1,   3,   2,   2,   3,   1,  10,
    0,-8, -45,   1,   1,   1,   1, -45,  -8,
    0,100,  -8,  10,   5,   5,  10,  -8, 100,
    0,0,0,0,0,0,0,0,0
};
```

角位我们给出了 100 的高分，与之对应的是星位，我们给出–45 的惩罚性价值分，C 位也是负价值分，目的是降低位置评估分，使得搜索算法避免落子到这些位置。有了这个表，计算棋子位置价值的算法就非常简单了，遍历、求和即可。

```
int GameState::CountPosValue(int player_id)
{
    int value = 0;
    for(int i = 0; i < GAME_CELLS; i++)
    {
        if(m_board[i] == player_id)
        {
            value += posValue[i];
        }
    }

    return value;
}
```


最后是完整的估值函数实现，其中的系数并不是最优值，最好多找一些棋局进行估值计算，并根据反馈调整这些系数，以期获得更好的效果。

```
int WzEvaluator::Evaluate(GameState& state, int max_player_id)
```

```
{
    int min =  GetPeerPlayer(max_player_id);
    int empty = state.CountEmptyCells();

    int ev = 0;
    if(empty >= 40) /*只考虑行动力*/
    {
        ev += (state.CountMobility(max_player_id) - state.CountMobility(min)) * 7;
    }
    else if((empty >= 18) && (empty < 40))
    {
        ev += (state.CountPosValue(max_player_id) - state.CountPosValue(min)) * 2;
        ev += (state.CountMobility(max_player_id) - state.CountMobility(min)) * 7;
    }
    else
    {
        ev += (state.CountPosValue(max_player_id) - state.CountPosValue(min)) * 2;
        ev += (state.CountMobility(max_player_id) - state.CountMobility(min)) * 7;
        ev += (state.CountCell(max_player_id) - state.CountCell(min)) * 2;
    }

    return ev;
}
```

23.3.3 搜索算法实现

23.2 节介绍井字棋游戏时提到的极大极小值搜索算法、带“α-β”剪枝的搜索算法和负极大值搜索算法的实现，本节我们再实现一种搜索算法，就是 23.1.4 节介绍的带“α-β”剪枝的负极大值搜索算法。由于黑白棋游戏搜索过程中状态极多，为了提高搜索效率，本节介绍的黑白棋游戏的搜索算法还采用了启发式搜索和置换表技术。再次重申一遍，和本书其他章节给出的算法示例一样，本节给出的算法都使用了最简单的实现形式，目的是为了让大家理解算法实现的原理，并不是要实现一个棋力超强的 AI。有时候为了算法的简洁会舍弃一些效率，比如接下来要介绍的置换表就使用了 STL 库的 map 容器，更高效的做法可以参考各种开源软件给出的解决方案。

1. 走法生成

根据黑白棋的规则，任何一方落子必需要能翻转对方的棋子，这就是黑白棋走法生成的唯一规则。根据这个规则，黑白棋的走法生成就是遍历当前的空位链表，对每个空位判断如果落子后是否能在 8 个方向中的任一个方向翻转对方的棋子：

```
int GameState::FindMoves(int player_id, int opp_player_id, std::vector<MOVES_LIST>& moves)
{
    std::vector<int> flips;
    MOVES_LIST ml;

    moves.clear();
    for(EMPTY_LIST *em = m_EmHead.succ; em != NULL; em = em->succ)
    {
        int cell = em->cell;
```

```
        int flipped = DoFlips(cell, player_id, opp_player_id, flips);
        if(flipped > 0)
        {
            m_board[cell] = player_id;
            em->pred->succ=em->succ;//cell 链表的 succ 链暂时跳过 em( CountMobility 函数会用到这个链表 )
            ml.goodness = -CountMobility(opp_player_id);
            em->pred->succ = em; //cell 链表的 succ 链恢复 em
            ml.em = em;
            UndoFlips(flips, opp_player_id);
            m_board[cell] = PLAYER_NULL;

            moves.push_back(ml);
        }
    }

    return moves.size();
}
```

for 循环遍历空位链表，DoFlips()函数尝试翻转对手的棋子（opp_player_id），如果返回值大于 0，则说明此处落子能翻转对方的棋子，被翻转的棋子位置被记录在 flips 数组中，因为在对下一个空位进行尝试之前，要调用 UndoFlips()函数将被翻转的棋子恢复。FindMoves()函数最后将合法的走法存入 moves 数组返回，你可能已经注意到了，moves 数组除了记录可落子的空位链表节点之外，还记录了一个 goodness，这个 goodness 就是如果在此空位落子能获得的好处，它记录的是落子后对手的行动力的负数，说明如果对手的行动力越大，这个落子获得的好处越低。记录这个值的目的是为后续启发式搜索提供启发依据，后面我们将介绍如何利用这个值进行启发搜索。

2. 引入置换表

黑白棋游戏搜索过程中会出现很多中间棋局状态，即使使用了“α-β”剪枝，中局时一个棋局的搜索还是可能超过 20 万个棋局状态（搜索深度 6 层）。这中间显然有很多棋局状态会重复出现，为此我们为搜索算法引入了置换表，希望通过置换表减少一些重复的搜索。

置换表的关键是查找和存储，高效的哈希算法是置换表技术必不可少的部分。很多棋类游戏都选择 Zobrist 哈希算法，原因在于 Zobrist 哈希算法简单，并且可以根据棋盘上少数位置的变化小范围地更新棋局的哈希值，不必因为改动几个棋子就全部重新计算一个棋局的哈希值，非常适合棋类游戏。本书的第 20 章介绍华容道游戏时已经介绍了 Zobrist 哈希算法的原理和实现，本章就不再重复说明。Zobrist 哈希算法需要为每个棋盘格子的状态准备一个随机数，因此需要根据棋类游戏中每个棋盘格子的状态多少进行调整，大家可以通过查看 othello 项目中的 InitZobristHashTbl()函数的源代码了解这种变化。

置换表的更新策略我们采用深度优先策略。为了防止置换表被搜索深度很深的棋局占满，无法保证棋局评估结果的实时性，我们选择在每次开始搜索前重置一下置换表，即在 SearchBestPlay()函数中调用 ResetTranspositionTable()函数。借助于 STL 的 std::map 容器的便利接口，置换表的查找和更新算法可以非常简单地实现：

```
bool LookupTranspositionTable(unsigned int hash, TT_ENTRY& ttEntry)
{
    std::map<unsigned int, TT_ENTRY>::iterator it = tt_map.find(hash);
    if(it != tt_map.end())
    {
        ttEntry = it->second;
        return true;
    }

    return false;
}

void StoreTranspositionTable(unsigned int hash, TT_ENTRY& ttEntry)
{
    std::map<unsigned int, TT_ENTRY>::iterator it = tt_map.find(hash);
    if(it != tt_map.end())
    {
        TT_ENTRY& old_entry = it->second;
        if(ttEntry.depth >= old_entry.depth)
        {
            old_entry = ttEntry;
        }
    }
    else
    {
        tt_map[hash] = ttEntry;
    }
}
```

3. 启发式搜索

调用 FindMoves()函数后得到一个所有合法走法的列表，在后续的博弈树搜索过程中，如果对每个合法的走法不假思索、机械地搜索，最终的结果就是盲目搜索。如果能利用一些额外信息对所有的走法进行适当的处理，减少一些无谓的搜索，即可称为启发式搜索。从这个角度理解，我们使用的“α-β”剪枝和置换表技术也是一种启发式搜索。事实上，棋类游戏中还有很多其他的启发因素，比如如果某个走法能吃掉对方的棋子，则优先对这个走法进行搜索，把不能吃子的走法放在后面搜索。如果有多个能吃子的走法，就根据吃掉的对方棋子的棋力从大到小排序，这种方法可以笼统地称为走法排序启发。

我们计划在黑白棋的搜索算法中应用一下走法排序启发。根据什么排序呢？前面介绍FindMoves()函数时统计了每一种走法的好处 goodness，实际上就是对手行动力的负值，我们就根据这个排序。因为我们的估值函数算法考虑了行动力因素，因此按照对自己有利的因素排序，首先搜索对自己最有利的走法，可以使得搜索算法能够更快地建立准确的剪枝窗口[α, β]，使后续的剪枝操作更高效。SortMoves()函数负责对走法数组排序，搜索算法每次调用 FindMoves()函数得到走法数组后，首先调用 SortMoves()函数排序，然后再开始具体的搜索操作。

4. 搜索算法实现

本节实现了一个带“α-β”剪枝的负极大值搜索算法 NegamaxAlphaBetaSearcher，同时引入

了置换表技术。完整的搜索算法在 NegaMax()函数中：

```
int NegamaxAlphaBetaSearcher::NegaMax(GameState& state, int depth, int alpha, int beta, int
    max_player_id)
{
    int alphaOrig = alpha;

    unsigned int state_hash = state.GetZobristHash();

    //查询置换表
    TT_ENTRY ttEntry = { 0 };
    if(LookupTranspositionTable(state_hash, ttEntry) && (ttEntry.depth >= depth))
    {
        if(ttEntry.flag == TT_FLAG_EXACT)
            return ttEntry.value;
        else if(ttEntry.flag == TT_FLAG_LOWERBOUND)
            alpha = std::max(alpha, ttEntry.value);
        else// if(ttEntry.flag == TT_FLAG_UPPERBOUND)
            beta = std::min(beta, ttEntry.value);

        if(beta <= alpha)
            return ttEntry.value;
    }

    if(state.IsGameOver() || (depth == 0))
    {
        return EvaluateNegaMax(state, max_player_id);
    }

    int score = -INFINITY;
    int player_id = state.GetCurrentPlayer();
    int opp_player_id = GetPeerPlayer(player_id);

    std::vector<MOVES_LIST> moves;
    int mc = state.FindMoves(player_id, opp_player_id, moves);
    if(mc != 0)
    {
        SortMoves(moves);

        std::vector<int> flips;
        for(int i = 0; i < mc; i++)
        {
            state.DoPutChess(moves[i].em, player_id, flips);
            state.SwitchPlayer();
            int value = -NegaMax(state, depth - 1, -beta, -alpha, max_player_id);
            state.UndoPutChess(moves[i].em, player_id, flips);
            state.SwitchPlayer();
            score = std::max(score, value);
            alpha = std::max(alpha, value);
            if(beta <= alpha)
                break;
        }
    }
    else
```

```
    {
        state.SwitchPlayer();
        score = -NegaMax(state, depth - 1, -beta, -alpha, max_player_id);
        state.SwitchPlayer();
    }

    //写入置换表
    ttEntry.value = score;
    if(score <= alphaOrig)
        ttEntry.flag = TT_FLAG_UPPERBOUND;
    else if(score >= beta)
        ttEntry.flag = TT_FLAG_LOWERBOUND;
    else
        ttEntry.flag = TT_FLAG_EXACT;

    ttEntry.depth = depth;
    StoreTranspositionTable(state_hash, ttEntry);

    return score;
}
```

在棋局搜索之前，首先调用 LookupTranspositionTable()函数查找置换表，如果置换表中存在搜索深度大于或等于当前搜索深度的结果，就直接使用这个结果。根据 flag 标志的值，可以分三种情况使用这个置换表条目：如果 flag 的值是 TT_FLAG_EXACT，则 value 的值就是最终估值，如果 flag 的值是 TT_FLAG_UPPERBOUND 或 TT_FLAG_LOWERBOUND，则 vlaue 的是目前对这个棋局搜索过程中已知的极大 α 和极小 β 值，在其后的搜索过程中可以据此更新当前的剪枝窗口[α, β]。

NegaMax()函数的最后阶段是将搜索结果存入置换表。存入置换表之前首先要根据当前搜索的结果更新剪枝窗口[α, β]的范围，如果当前搜索的估值不在这个范围，则说明这个估值是个精确值，需要设置 flag 标志的值为 TT_FLAG_EXACT。如果当前估值小于 α，则说明当前估值可作为后续搜索的极大剪枝（α 剪枝）边界值。如果当前估值大于 β，则说明当前估值可作为后续搜索的极小剪枝（β 剪枝）边界值。

23.3.4 最终结果

至此，与黑白棋游戏相关的棋盘模型、估值函数、搜索算法都介绍完了，将以上算法实现放入我们的棋类游戏代码框架，就可以得到一个控制台界面的黑白棋对战程序。我们的 AI 算法棋力虽然不高，但是我仍然不能战胜它，搜索深度是 6 层的情况下，我被电脑杀得一败涂地。我用一个搜索深度是 4 层的电脑和一个搜索深度是 3 层的电脑对战，发现搜索深度是 4 层的电脑几乎 100%地获胜，看来基于博弈树搜索的 AI 算法，能多搜索一层就能占很大的优势。网上公开的几个棋力比较强的几个 AI 算法，在终局阶段都能达到 18 ~ 22 层的搜索深度，几乎能搜到最终状态了。

23.4 五子棋

五子棋流行非常广泛，在不同的国家有不同的名称，英文名称为 FIR（Five In a Row）。标准的五子棋棋盘是 15×15 大小，用数字 1～15 标识棋盘的行，用字母 A～O 标识棋盘的列，棋子和围棋一样有黑白两种颜色，可以和围棋的棋局通用。下棋的双方轮流在 15×15 条线的交叉点上落子，先在横、竖和斜线方向上形成五子连线的一方获胜。

作为一种策略类的游戏，五子棋也有很多"型"，按照五子棋的术语称为"冲四""活三"等。首先来介绍一下"冲"，棋型的两端有界的棋称为"冲"，根据相连棋子个数可有"冲二""冲三""冲四"等说法。图 23-7 就是黑棋"冲四"的两种棋型，与黑棋的一端直接接触的要么是白棋，要么是边界，如果黑棋的两端都是空位，则这个棋型就成了"活四"。"活"的定义是棋型的两端都是无界约束（两端不和对手的棋子或边界直接接触），但是根据相连棋子的个数，对两端的空位的数量也有要求。以图 23-8a 所示的"活三"为例，除了要求两端为空位外，还要求其中一端至少有两个空位，即要求空位数至少有三个。图 23-8b 所示的"跳活三"是另一种情况，加上中间空位也是至少需要三个空位。

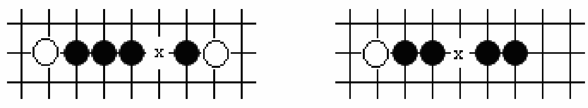
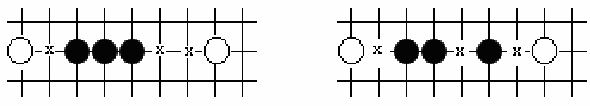


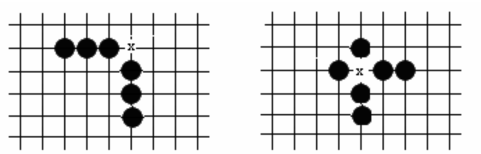
图 23-7 "冲四"棋型示意图



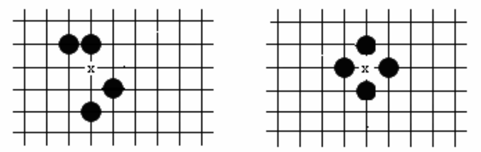
(a) (b)

图 23-8 "活三"棋型示意图

大多数的棋类游戏先手落子一方都会不同程度地占有一些优势，五子棋也不例外。不仅如此，现代计算机的大量模拟计算证明，五子棋存在一些特定的走法，按照步骤走这些走法可以保证能战胜对手。为此，人们设置了很多五子棋特有的比赛规则，"禁手"就是其中一种最常用的方法。所谓"禁手"，就是禁止先手一方（通常是黑棋）走某些特定的棋型，这些棋型要么会使得先手一方占有某种不平等优势，要么会使得先手一方必胜。"禁手"棋型有很多，比如常见的"四四禁手""三三禁手""长连禁手"等。图 23-9a 是"四四禁手"的两种常见棋型，图 23-9b 是"三三禁手"的两种常见棋型。一般来说，比赛中黑棋只要走了"禁手"，白棋可立即指出，此时判黑棋负。如果白棋没有指出，则比赛继续进行。在某些情况下，如果规则允许，白棋甚至可以逼迫或诱骗黑棋走出"禁手"，从而赢得比赛。当然，黑棋如果看出白棋的阴谋，但是又无其他路可走，还可以选择放弃一手，也就是让白棋再走一步，无论如何，这对黑棋都非常不利。



(a)



(b)

图 23-9 “禁手”棋型示意图

根据以上对五子棋游戏规则的分析，可知一个五子棋游戏的 AI 算法除了搜索算法、估值算法等内容之外，还要能识别出特殊的棋型，比如各种“冲”或“活”的棋型，这些都是估值函数评估的依据。除此之外，还要能识别出各种“禁手”棋型，并且在走法生成的时候直接过滤掉“禁手”。接下来我们将介绍五子棋游戏的数据模型，这个模型的好坏将直接影响到棋型判断算法的实现复杂度。

23.4.1 棋盘与棋子的数学模型

23.3.1 节介绍黑白棋游戏的棋盘数据模型时，介绍了 Warren Smith 在其黑白棋终局处理算法中使用的一种棋盘状态模型（如下所示），我们的黑白棋 AI 算法也使用了这个模型，现在我们的五子棋游戏也继续使用这个模型。

```
dddddddddd
dxxxxxxxxx  11
dxxxxxxxxx  21
dxxxxxxxxx  31
dxxxxxxxxx  41
dxxxxxxxxx  51
dxxxxxxxxx  61
dxxxxxxxxx  71
dxxxxxxxxx  81
dxxxxxxxxx  91
ddddddddddd
```

关于直接使用二维数组和使用一维数组的优缺点在 23.3.1 节已经介绍过了，这里不再赘述。五子棋游戏的棋盘和黑白棋游戏的棋盘有很大的差异，需要对 Warren Smith 的模型做适当的修改。标准的五子棋游戏是 15×15 的棋盘，但是我们演示 AI 算法的程序使用 9×9 的小棋盘，一方面是为了便于展示算法的实现效果，另一方面是加快计算机“想棋”的速度，毕竟棋盘减小了，需要的计算量会呈几何级数减少。9×9 的小棋盘用 Warren Smith 模型表示，需要一个长度为 111

的一维数组表示黑白棋的棋盘与棋子状态，其中 81 个是棋盘上的位置，30 个是标志位或哨兵位。111 个数组元素中前 11 个和后 11 个是标志位，中间每间隔 9 个 *x* 位置插入一个标志位 d，这个模型各个位置的逻辑结构如上一页的阵列所示。

棋盘大小调整了，方向数组的步进量也需要调整。五子棋的搜索方向比较简单，只关注棋子在横、竖和两条斜交叉线上是否有连续出现的情况，只需沿四个方向搜索即可。根据 Warren Smith 模型的关系，适用于 9×9 棋盘的方向数组调整如下：

```
const int dir_inc[] = {1, 9, 11, 10};
```

模型中的元素与实际棋盘上的行和列的坐标换算关系也调整为：

```
square(row,col) = board[11+col+row×10]   （0<= row,col <=8）
```

至此，整个五子棋的数据模型就建立了。

应用这个一维棋盘模型，可以极大地方便后续算法的设计，现在就以判断是否有棋手完成五子连珠的算法为例，演示一下这个模型给我们的算法实现带来的便利。下棋的双方每落下一子，游戏控制器就要检查棋盘状态上是否构成了五子连珠，如果有五子连珠，则设置游戏结束标志，并给出胜利者的 ID 以便最后输出胜负结果。这个检查算法的原理很简单，就是从落子位置开始，在一条线上沿正向和反向分别搜索与落子棋子相同的棋子个数，如果从正、反两个方向搜索到的相同棋子个数之和大于或等于 5，则判定有棋手完成了五子连珠。

```
bool GameState::CheckLinefive(int cell, int dir_inc, int player_id)
{
    int count = 1;
    int ct = cell - dir_inc;
    while(m_board[ct] == player_id)
    {
        count++;
        ct -= dir_inc;
    }

    ct = cell + dir_inc;
    while(m_board[ct] == player_id)
    {
        count++;
        ct += dir_inc;
    }

    return (count >= 5);
}
```

CheckLinefive()函数每次搜索一条线，步进增量 dir_inc 取负表示沿者这条线的向反方向搜索。对四条线都搜索一次就可以判断在四个方向上是否有五子连珠，这正是 CheckFiveInRow()函数所做的事情。


```
bool GameState::CheckFiveInRow(int cell, int player_id)
{
    for(int i = 0; i < DIR_COUNT; i++)
```

```
    {
        if(CheckLinefive(cell, dir_inc[i], player_id))
        {
            return true;
        }
    }

    return false;
}
```

23.4.2 估值函数与估值算法

五子棋游戏的局面评估主要是根据棋型来评估，比如己方棋子达成五子连珠，表示这是一个胜局，应该给出最高分，如果是对手的棋子达成五子连珠，表示这是最糟糕的局面，就应该给出最低分。如果有“活四”出现，就意味着离胜利只有一步之遥，也应该给予适当的评估分。由此可见，正确地识别出五子棋的各种棋型是五子棋估值算法实现的关键。

1. 棋型计算

五子棋的棋盘上，单个的棋子对对手基本上没有太大的威胁，两个以上的连子才开始对对手构成威胁，因此棋型的识别应该从“活二”和“冲二”开始。“冲三”和“活三”的威胁就又进了一步，特别是“活三”，如果对手此时不及时处置，再过一手就发展成“活四”，这是必胜的棋型之一。

五子棋的棋型识别是基于横线、竖线和正反两条斜线共四个方向，横线和竖线比较规整，但是正反两条斜线比较难处理，这正是我们放弃二维棋盘模型的原因。根据我们的数据模型定义，我们将线定义为一个起点和一个方向步进量组成的二元组，起点是棋盘上的点对应的数据模型中的位置（一维数组的下标），从起点开始，通过叠加步进量移动到下一个点，逐次叠加步进量直到遇到哨兵位，这期间的点就是这条线上的点。图 23-10 是棋盘上的线与数据模型的位置关系示意图，图中每个圆圈代表 9×9 棋盘上的一个位置，圆圈中的数字是这个棋盘位置在数据模型中的位置。从图中可以看到，九条横线的起点分别是 11、21、31、41、51、61、71、81、91 这九个点，其方向步进量是 1。九条竖线的起点分别是 11、12、13、14、15、16、17、18、19 这九个点，其方向步进量是 10。斜线需要注意一下，四个角上的斜线如果棋子总数小于 5 是可以排除掉的，因为这些线上肯定构不成五子连珠。正斜线方向的起点是 11、12、13、14、15、21、31、41、51 这九个点，其方向步进量是 11。同样，反斜线方向的起点是 15、16、17、18、19、29、39、49、59 这九个点，其方向步进量是 9。

按照这个思路，我们先给出线的定义：

```
typedef struct tagLines
{
    int line_s[MAX_LINE_S];
    int off_dir;
}LINES;
```

然后根据图 23-10 准备好四个方向上所有线的起点列表和方向步进量：

```
LINES line_cpts[4] = {…}
```

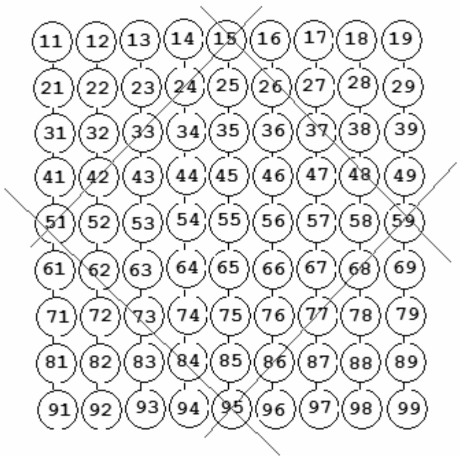


图 23-10 棋盘线与数据模型位置关系示意图

SearchPatterns()函数的外层 for 循环完成对四个方向的遍历，内层 for 循环完成对每个方向上 9 条线的遍历，ev_ata 参数返回棋型识别的结果。所有的线都可以用 AnalysisLine()函数进行一致化处理，这就是我们的数据模型的优点。

```
void GameState::SearchPatterns(EvaluatorData &ev_ata)
{
    for(int i = 0; i < COUNT_OF(line_cpts); i++)//每个方向
    {
        for(int j = 0; j < MAX_LINE_S; j++)//每个方向条线
        {
            AnalysisLine(line_cpts[i].line_s[j],
                         line_cpts[i].off_dir, ev_ata);
        }
    }
}
```

AnalysisLine()函数不关心是横线还是竖线，它只根据线的起点和方向步进量进行扫描，可以一次性将一条线上黑白棋的棋型都识别出来。棋型识别的关键是先找出连续棋子的开始位置和结束位置，然后在这基础上向前和向后寻找空位，如果连子数和空位数小于 5，则说明这些连子最后不可能形成五子连珠，不会构成威胁，统计时可忽略这些连子，避免影响估值算法的结果。只有连子数和空位数大于 5 的时候，连子才可能对对手构成威胁，此时需要进一步判断是“冲”还是“活”。如果连子的两端都有空位，且任意一端的空位数大于或等于（5 – 连子数 – 1），则直接判定为“活”。如果连子的两端任意一端是边界或对方的棋子，则判定为“冲”，如果连子两端是空位，但是不满足“活”条件的也判定为“冲”。以上就是识别算法的简单描述，具体实现请看 AnalysisLine()函数的代码：

```
void GameState::AnalysisLine(int st, int dir_inc, EvaluatorData &ev_ata)
{
    int mark_cell,mark_player_id;
    int ct = st;
    while(m_board[ct] != DUMMY)
    {
        ct = SkipEmptyCell(ct, dir_inc);//向后跳过空位
        if(m_board[ct] == DUMMY) //已经到哨兵位？直接结束
            break;

        mark_cell = ct;
        mark_player_id = m_board[ct];
        int count = 0;
        ct = SearchAndCountChess(ct, dir_inc, mark_player_id, count);
        if(count >= 5)
        {
            ev_ata.IncreaseCounter(5, mark_player_id, false);
        }
        else if(count >= 2)
        {
            int pre_space = 0;
            int succ_space = 0;
            //向前寻找空位
            int tmp_t = mark_cell - dir_inc;
            tmp_t = SearchAndCountChess(tmp_t, -dir_inc, PLAYER_NULL, pre_space);
            //向后寻找空位
            ct = SearchAndCountChess(ct, dir_inc, PLAYER_NULL, succ_space);
            if((m_board[ct] == mark_player_id) && (succ_space == 1))
            {
                //处理“跳”的情况
                count++; //多了一个棋子
                int space_need = 5 - count;
                bool succ_close = (m_board[ct + dir_inc] != PLAYER_NULL);
                if((pre_space + succ_space) >= space_need)
                {
                    ev_ata.IncreaseCounter(count, mark_player_id, succ_close);
                }
            }
            else
            {
                //除了 count 个连子之外，还需要 5-count 个空位，才能构成冲 x 或活 x
                int space_need = 5 - count;
                //两端都有空位，且任意一端的空位数大于等于 space_need，直接定为活 x
                if( ((pre_space > 0) && (succ_space > 0))
                    && ((pre_space >= space_need) || (succ_space >= space_need)) )
                {
                    ev_ata.IncreaseCounter(count, mark_player_id, false);
                }
                else
                {
                    //两端是否有封闭
                    bool pre_close = (m_board[mark_cell - dir_inc] != PLAYER_NULL);
                    bool succ_close = (m_board[ct] != PLAYER_NULL);
                    //空位足够连成 5 子才统计
```

```
                    if((pre_space + succ_space) >= space_need)
                    {
                        ev_ata.IncreaseCounter(count, mark_player_id, pre_close||succ_close);
                    }
                }
            }
        }
    }
}
```

SearchAndCountChess()函数从 cs 参数指定的开始位置搜索指定 id 的棋子的个数（通过 count 参数返回），返回搜索结束时的位置。尽管通过 SearchAndCountChess()函数减少了几十行代码，但是 AnalysisLine()函数仍然是本书迄今为止最长的函数，不过我相信你一定见过比这更长的棋型识别算法。

2. 估值算法

对棋局进行估值，除了识别出棋型，还要给不同的棋型指定评估分数，以便估值函数进行计算。本书给出了一种简单的棋型记分规则：

- 五子连珠计 10000 分
- “活四”“双冲四”“冲四活三”这三种情况分别计 9900 分
- “双活三”“双冲四”这两种情况分别计 9800 分
- “活三冲三”“冲四活三”这两种情况分别计 9700 分
- “冲三”一次计 300 分
- “活二”一次计 200 分
- “冲二”一次计 50 分

除此之外，我们的估值算法还考虑位置分。对于五子棋游戏来说，边是比较差的位置，靠近边的一侧发展受限，除非迫不得已或谦让对手，一般情况下棋手都不会先靠边上落子。但是计算机傻，特别是在开局阶段，棋盘上的子很少，棋型的估值贡献为 0，此时计算机就会随机落子，有可能就落在边上。为了告诉计算机在这种情况下如何处理，我们给棋盘的每个点设置了位置分。边界上的点位置分是 0，越靠中间位置分越高，告诉电脑如果不知道怎么落子的时候，就往中间位置放。

评估分超过 9000 的都是必胜的棋局，这种情况下就根据棋手的情况直接返回分数，其他情况下统计包括位置分在内的棋型得分。在前面介绍的棋型计算的基础上，评估算法的实现就非常简单了，此处就不再列出代码。

23.4.3　搜索算法实现

搜索算法我们依然采用带“α-β”剪枝的负极大值搜索算法，这个算法在 23.3 节介绍黑白棋的时候已经介绍过。本节将介绍与五子棋有关的走法生成和“禁手”判断。

1. 走法生成

除了"禁手"之外，五子棋的落子没有特殊的规则，棋盘上任意空位都可以落子。因此走法生成算法就是遍历棋盘上的所有空位，排除掉"禁手"位置，剩下的就是可走的位置。当然，我们也可以根据位置分对所有可落子的位置进行排序，作为走法排序启发搜索的依据。走法生成由 FindMoves()函数实现，因为算法简单，这里也不列出代码了。

2. "禁手"判断

"禁手"是一种特殊的棋型，如果将"禁手"理解为在一个可以落子的位置周围几个特定位置上不能同时有己方棋子，那么对"禁手"建模就非常简单了。以图 23-9 的"禁手"示意图为例，如果黑棋想在 x 位置落子，需要判断在几个黑棋位置是否都有黑棋，这些位置与 x 位置存在某种关系，根据我们的棋盘数据模型，这种关系就是方向步进偏移。以图 23-9a 左侧的"四四禁手"示意图为例，x 位置左侧的三个黑棋位置与 x 的方向步进偏移分别是–1、–2 和–3，x 位置下方的三个黑棋位置与 x 的方向步进偏移分别是 10、20 和 30。

根据以上分析，我们将"禁手"的数据模型定义为：

```
typedef struct tagForbiddenItem
{
    int off_inc[MAX_FORBIDDEN_PATTERN];
    int off_cnt;
}FORBIDDEN_ITEM;
```

off_cnt 记录这个"禁手"棋型中相关的棋子个数，off_inc 数组记录这些棋子相对当前位置的方向步进偏移。利用这个数据模型，预先将各种"禁手"组织成一个列表：

```
FORBIDDEN_ITEM forbidden_patterns[] =
{
    {
        { -1,-2,-3,10,20,30 },
        6
    },
    ...
};
```

"禁手"判断的算法就是遍历这个"禁手"表，对每个"禁手"模型判断相关位置上的己方棋子是否与"禁手"模型匹配，如果匹配则说明当期落子位置是一个"禁手"。对单个"禁手"模型匹配的算法实现如下：

```
bool GameState::IsMatchSingleForbidden(FORBIDDEN_ITEM& item, int cell, int player_id)
{
    int match_cnt = 0;
    for(int j = 0; j < item.off_cnt; j++)
    {
        int cf = cell + item.off_inc[j];
        if((cf >= 0) && (cf < BOARD_CELLS))
        {
            match_cnt += ((m_board[cf] == player_id) ? 1 : 0);
        }
```

```
        }

        return (match_cnt == item.off_cnt);
    }
```

match_cnt 记录匹配的棋子个数，如果 match_cnt 与这个模型中的棋子个数相等，则说明符合该“禁手”模型，是一个禁手。

23.4.4 最终结果

将以上算法实现放入我们的棋类游戏代码框架，就可以得到一个控制台界面的五子棋对战程序。这个 AI 还是比较弱智的，看来还有很大的改进余地。如果要实现一个 15 × 15 标准棋盘的五子棋游戏，只需修改几个常量定义和 GameState 类的几个数据结构即可，整个算法都是通用的。只是换成标准棋盘后计算量太大，搜索深度为 3 的时候计算机每一次要想很长时间，你要有心理准备。

23.5 总结

博弈树的搜索是当前棋类游戏的 AI 基础算法，当某一天计算机的处理能力强到可以搜索出所有棋局状态的时候，计算机之间的对战就真的是一点意思都没有了，有的棋是先行者总是胜利，有的棋则无论如何都是平局。博弈树的搜索不仅仅用于棋类游戏，它是人工智能领域一个重要的研究方向，许多完全信息的二人零和博弈问题都可以用博弈树搜索算法解决。在博弈树搜索算法方面，前人做了许多丰富而充满意义的研究工作，这些都是我们研究这些算法乐趣的来源。

本章介绍了几种最基本的博弈树搜索算法和三种简单的棋类游戏实现，我并不是这些棋类游戏的高手，所以不要指望我“调教”出来的算法有太高的“智商”。但是本章实现的算法都是实现一个自动下棋的 AI 的基本内容，可以作为继续提高“智商”的起点。改进可从几个方面进行，首先是数据模型的改进，可以使用数据量更小的“位棋盘”（bitboard），将棋盘状态的数据减少到 128 比特以内，就可以充分利用现代 CPU 的高阶寄存器提高计算和数据处理的速度。其次是搜索算法的改进，比如应用剪枝效率更高的 PVS 或 MTD(f)算法，启用开局库和更高效的启发搜索等。最后是估值函数的设计，本章给出的都是最基本的估值函数，某些系数都不是最优的，所以棋力不强，有很大的改进余地。改进估值函数的算法，是提高棋力最直接的方法。

最后再啰唆一下，看似复杂和神秘的东西，只要有简单的理论指导，其实现一定也简单。棋类游戏的 AI 算法再次印证了这一点，以后对电脑下棋应该不会再感到神奇了，基本的东西就是这些，关键就是细节的处理，谁的细节处理得强大，谁的棋力就强大。

23.6 参考资料

[1] Cormen T H, et al. *Introduction to Algorithms* (*Second Edition*). The MIT Press, 2001

[2] 周伟中. 棋类游戏 100 种. 北京：人民体育出版社，2009

[3] 王小春. PC 游戏编程——人机博弈. 重庆：重庆大学出版社，2002

[4] Allis V. Searching for Solutions in Games and Artificial Intelligence. PhD thesis, Department of Computer Science, University of Limburg, 1994

[5] Knuth D E, Moore R W. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6:293－326, 1975

[6] 维基百科 "极大极小值算法"：http://en.wikipedia.org/wiki/Minimax

[7] 维基百科： http://en.wikipedia.org/wiki/Tic-tac-toe

[8] 维基百科 "负极大值算法"： http://en.wikipedia.org/wiki/Negamax

[9] 维基百科"Alpha-beta 剪枝"： http://en.wikipedia.org/wiki/Alpha-beta_pruning

[10] Marsland T A. A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9 No. 1, 3-19, ISSN 0920-234X ,1989

[11] Schaeffer J. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-11, No.11, 1203-1212, 1989

[12] 维基百科 "置换表"： http://en.wikipedia.org/wiki/Transposition_table

[13] Edwards D J, Hart T P. The Alpha－Beta Heuristic (AIM-030). Massachusetts Institute of Technology, 2006

[14] Marsland T A. *The Anatomy of Chess Programs*. ICCA President, 1992-1999

[15] Marsland T A, Schaeffer J. *Computers, Chess, and Cognition*. Springer-Verlag, 1990

[16] Laramée F D. *Chess Programming*. *Artificial Intelligence*, 2000

[17] 游戏开发者网站：http://www.gamedev.net

[18] Pearl J. *Heuristic-Intelligence Search Strategies for Computer Problem Solving*. 1984

[19] 维基百科：http://zh.wikipedia.org/wiki/黑白棋

[20] http://www.radagast.se/othello/endgame.c

[21] Mandt M. Introduction to Basic Othello Strategy and Algorithms. *ICCA Journal*, 2001

[22] Andersson G. Writing an Othello program. www.radagast.se /Othello, 2007

[23] Buro M. ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm. *ICCA Journal* 18(2), 1995

[24] Buro M. The Evolution of Strong Othello Programs. Proceedings of the IWEC-2002 Workshop on Entertainment Computing, 2002

[25] Buro M. Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello. *Artificial Intelligence*, 1995

[26] 维基百科 "五子棋"：http://zh.wikipedia.org/wiki/Renju

附录 A 算法设计的常用技巧

实现一个算法的目的是解决一个实际的问题，编写一个解决问题的算法涉及数据结构、问题域内相关的理论知识、对程序构造的理解等方面的知识。当然，具体到算法实现阶段，也有很多实用的小技巧可以起到锦上添花的作用。本书整理了一些编写算法实现常用的技巧，希望这些技巧在你写程序的时候能派上用场。

A.1 数组下标处理

数组的下标是一个隐含的很有用的属性，巧妙地使用这个属性，对简化算法实现有很大的帮助。本书 2.2.1 节介绍的统计数组中重复元素个数的例子就使用了这个技巧，只用两行代码就实现了这个算法。在 4.4.2 节介绍阿拉伯数字转中文数字的例子中再次使用了这个技巧，结合一个预先准备好的中文数字与阿拉伯数字对照表：

```
const char *chnNumChar[CHN_NUM_CHAR_COUNT] = { "零", "一", "二", "三", "四", "五", "六", "七", "八", "九" };
```

利用数组下标只需一行代码就可找到阿拉伯数字对应的中文数字，比如数字 5 对应的中文数字就是：

```
chnNumChar[5]
```

在某些情况下，问题域内的一些特殊数据元素，比如 ID、类型等标识性属性，如果能定义成从 0 开始的连续整数，也可以利用数组和数组下标的特殊关系，简化数据模型，优化代码结构。比如第 8 章介绍“爱因斯坦的思考题”解法时，就将房子颜色、国籍、饮料类型、宠物和香烟牌子作为类型属性，定义成从 0 开始的索引值（为保证可读性，定义成有意义的常量值）：

```
type_house = 0,
type_nation = 1,
type_drink = 2,
type_pet = 3,
type_cigaret = 4
```

然后将这五种类型属性定义成数组：

```
int itemValue[GROUPS_ITEMS];
```

现在要查看一个 GROUP 绑定组中房子的颜色是否是蓝色，就可以这样编写代码：

```
if(group.itemValue[type_house] == COLOR_BLUE)
```

这样的例子应用得非常广泛，只要控制好数组越界问题，巧妙地设计数据结构，定义有意义的常量名称，可以在不影响代码可读性的基础上极大地简化算法实现。

A.2 一重循环实现两重循环的功能

二维表的遍历一般需要两重循环来实现，但是两重循环的代码不如一重循环的代码清爽，很多情况下用一重循环遍历二维表也是一种不错的选择。用一重循环遍历二维表关键是对下标的处理，对于一个 M × N 的二维表，可用以下方法解出对应的二维下标：

```
int row = i / M
int col = i % N
```

反过来，也可以用以下公式将二维坐标还原为一维坐标：

```
int i = row * N + col
```

本书 2.1.2 节介绍循环结构时，就介绍了一个用一重循环初始化九宫格游戏棋盘的算法实现。

A.3 棋盘（迷宫）类算法方向遍历

棋盘或迷宫类游戏常常需要配合各种搜索算法，二维棋盘和迷宫的搜索常常是沿着与某个位置相临的 4 个或 8 个方向展开，对这些方向的遍历就是搜索算法的主要结构。我常常看到一些朋友给出的算法用了长长的 if-else 或 switch-case 语句，无非是这样的结构：

```
switch(direction)
{
    case UP:
    ...
    case DOWN:
    ...
    case LEFT:
    ...
    case RIGHT:
    ...
}
```

观察每一个 case 分支，除了数组下标计算不同，其他代码都是雷同的重复代码。其实这种情况下最常用的方法是使用方向偏移数组，用一个循环对这个方向数组遍历一遍就可完成对各个方向的搜索。以二维数组定义的棋盘为例，如果从 i 行 j 列开始向上、下、左、右 4 个方向搜索，则这 4 个方向可转换为以下行、列坐标关系。

- ❑ 向左搜索：行坐标 i 不变，列坐标 $j-1$
- ❑ 向上搜索：行坐标 $i-1$ 不变，列坐标不变
- ❑ 向右搜索：行坐标 i 不变，列坐标 $j+1$

❑ 向下搜索：行坐标 i+1 不变，列坐标不变

根据以上关系，首先定义二维数组下标偏移量：

```
typedef struct tagIdxOffset
{
    int row_offset;
    int col_offset;
}OFFSET;
```

然后定义一个偏移量数组，分别表示向 4 个方向的数组下标偏移量：

```
OFFSET dir_offset[] = {{0,-1},{-1,0},{0,1},{1,0}};
```

假设当前位置的二维数组下标是 row，col，则对此位置开始向 4 个方向搜索的代码可以如此实现：

```
for(int i = 0; i < count_of(dir_offset); i++)
{
    int cur_row = x + dir_offset[i]. row_offset;
    int cur_col = y + dir_offset[i]. col_offset;
    ...
}
```

这种算法实现避免了对每个方向都进行下标计算，即便是增加两个斜线方向，从 4 个方向搜索扩展到 8 个方向搜索，只需调整 dir_offset 数组即可，摆脱了冗长的 switch-case 代码结构。

本书的 14.5.1 节介绍填充算法的时候就使用了方向数组。第 20 章介绍华容道游戏的时候再次使用了方向数组，都是类似情况下的典型应用。

A.4　代码的一致性处理技巧

经常做测试的程序员都知道，数据操作的边界是最容易出错的地方，从代码实现的角度理解这个问题，是因为边界数据的处理往往和内部数据的处理不太一样。第 1 章介绍环形队列的时候我们就遇到了这样的问题，因为内存中没有真正的环形数据存储机制，因此我们设计的环形队列是个逻辑存储结构，内部是用数组做存储支撑。环形队列是无界的，但是数组是有界的，这就产生一个问题，tail 指针每次向后移动时，都要判断是否移动到数组的边界。在这个例子中，我们用以下方式解决了这个问题：

```
tail = (tail + 1) % N
```

当 tail 指针超过数组的下标时，这个对 N 取余的操作会让 tail 自动调整到数组的头部，避免了 if-else 特殊处理，这是个一致性处理的简单例子。

第 4 章在介绍阿拉伯数字与中文数字转换时提到中文节权位的一些规则，万以上的数字节权位是“万”，亿以上的数字节权位是“亿”，但是万以下的数字没有节权位，这就是个例外，代码中可能到处需要对这个例外进行处理。现在换个思路，给节权位定义一个索引，万以下索引为 0，万以上索引为 1，超过亿索引为 2，以此类推，这样就可以定义一个节权表：

```
const char *chnUnitSection[] = { "", "万", "亿", "万亿" };
```

在确定节权的时候根据节权位索引查这个表，代码按照一致的方法添加节权位：

```
chnString += chnUnitSection[widx];
```

没有节权位时得到一个空字符串，chnString 加一个空字符串不影响结果，就这样避免了对节权位的 if-else 判断。

本书第 23 章介绍 Tic-Tac-Toe 游戏时，还介绍了一种利用预先编制的数据表对某些操作进行一致性处理的方法。Tic-Tac-Toe 游戏需要判断是否有三点连成一线，检查的方向有横、竖和两条斜交叉线共 8 个方向。如果按照一般的处理方法，可能需要分别用四种数组下标处理方法才能完成对 8 个方向的处理。但是 23.2.1 节介绍了一个方法，将 8 个方向的数组下标预先存为一张数据表，检查三点一线的时候直接从这张数据表获取每个方向对应的数组下标，使用一个循环就完成了对 8 个方向的检查，这也是一致性处理的例子。

除了以上几个例子，很多算法还通过设置标志位来避免算法实现过程中频繁判断边界值的状态。本书的第 20 章和第 22 章在棋盘中设置的棋盘边界标志位，就是这种一致性处理的示例，通过边界值的一些特殊设置，避免了对棋盘边界值的特殊判断，相关的原理都已经在相关章节中做了具体的说明。这些标志位在一些算法中也称为“哨兵位”，比如插入排序算法，就在待排序的线性表的最前面放置一个比数列中最小的数还小的数作为哨兵位，在插入搜索过程中就不需要每次都判断线性表的下标是否移动到了表头。

在算法设计中巧妙地使用一致性处理，可以极大地减少算法实现的复杂度，写出短小精悍的算法实现。把算法代码写短一点的意义不仅是展示技巧，更重要的原因是臃肿的代码容易出错，越短的代码越不容易出错。在算法中使用一致性技巧，需要巧妙地设计算法，精心构造数据结构，必要时需要事先计算并构造一些数据表，没有定势的方法，只能在各种算法中体会。

A.5 链表和数组的配合使用

动态存储的线性表常常用链表表示，但是频繁的插入和删除操作会使得内存操作的压力很大，频繁地申请和释放内存严重影响算法效率。为此人们提出了很多将链表和数组的优点结合在一起的方法，比如数组链表，这个在本书第 2 章也提到过。第 23 章我们还介绍了另一种将数组和链表相结合的方法，就是黑白棋算法中用到的空位链表。黑白棋使用 8×8 的棋盘，棋盘上的空位最多就是 64 个，因此可以利用数组定义一次性分配好内存：

```
EMPTY_LIST m_Empty_s[64];
```

再定义一个头节点：

```
EMPTY_LIST m_EmHead;
```

最后再用相关的指针将它们串成链表：

```
void GameState::InitEmptyList()
{
```

```
    int k = 0;
    EMPTY_LIST *pt = &m_EmHead;
    for(int i = 0; i < BOARD_CELLS; i++)
    {
        if(m_board[i] == PLAYER_NULL)
        {
            pt->succ = &(m_Empty_s[k]);
            m_Empty_s[k].pred = pt;
            pt = pt->succ;
            pt->cell = i;
            k++;
        }
    }
    pt->succ = NULL;
}
```

使用这个链表就可以不用考虑内存的申请和释放，还可以通过直接遍历 m_Empty_s 数组了解各个节点的情况，这种方法在很多强调内存分配效率的开源软件中都有应用。

A.6　“以空间换时间”的常用技巧

“以空间换时间”也是算法设计中常用的提高算法效率的技巧，有时也可用于一致性处理技巧简化算法实现。我们来举个简单的例子，加入我们需要对 cell_info 按照掩码进行不同的处理：

```
for(int i = 0; i < 8; i++)
{
    unsigned char mask = 0x01 << i;
    if(cell_info & mask)
    {
        //do something
    }
}
```

mask 掩码每次都需要通过 0x01 移位得到，如果我们换个方式，事先计算好一个掩码表：

```
unsigned char mask_tbl[] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
```

然后就可以将代码修改为：

```
for(int i = 0; i < 8; i++)
{
    if(cell_info & mask_tbl[i])
    {
        //do something
    }
}
```

这就是“以空间换时间”。当然，这是一个微不足道的例子，但是这种策略在很多地方都得到了应用，特别是这种计算比较繁琐或耗时的时候，预先准备好这些结果，避免每次都计算就是一个很好的策略。本书第 7 章介绍的舞伴之间的偏爱关系表和第 23 章介绍的棋盘位置价值表都是这种“以空间换时间”策略的应用示例。

A.7 利用表驱动避免长长的 switch-case

函数表驱动是状态机设计常用的方法，拿来在一般场合下使用，有时候也可以起到非常好的代码优化效果。代码中出现常常的 switch-case 结构或 if-else 结构往往是代码冗余的表现，将这些分支的处理代码提炼成函数往往是解决问题的第一步，但是仅仅这样做还不够。接下来还要统一这些函数的接口，如果能提供一致的接口，则可将其转换成函数表，从而以一种更优雅的方式重构这段代码。本书 2.1.3 节介绍分支跳转结构时介绍了一个这样构造函数表的例子，具体的实现代码请参考我的博客。

以上是经典的重构书中都会介绍的代码优化方法，重点是需要提炼出一批函数。假如通过上述优化最后得到了一批接口一致的函数，却发现这些函数所做的事情几乎一模一样怎么办？这也是问题，因为有重复代码。现在反过来考虑这个问题，能否将条件抽象成一致的外观，从而用一个函数一致地处理这些条件呢？这也是表驱动的方法，本书第 6 章介绍的过河动作表列表，就使用了这种方法，将过河动作抽象成一组数据，然后用一个函数一致地处理所有的过河动作。

switch-case 结构或长长的 if-else 分支结构往往意味着代码违反了"开闭原则"，这样的代码结构往往会随着后期代码的维护和功能的增多不断增加新的 case 分支和 else 分支，使得代码对修改永远无法封闭。避免这种代码结构的方法有很多，除了本节介绍的提炼函数表的方法，第 3 节介绍的方法和第 4 节介绍的一致性处理技巧也是代替 switch-case 结构的常用方法。

附录 B
一个棋类游戏的设计框架

在本书的第 23 章介绍了几种基于博弈树搜索的棋类游戏 AI 算法设计，为了演示这一章介绍的三种棋类游戏的 AI 算法，我们设计了三个可以实现人机对战的简单棋类游戏。这三个游戏程序使用了一套相似的代码框架，第 23 章着重介绍博弈树的搜索算法设计，并没有介绍这套代码框架。如果读者想在本书的代码基础上进一步演化和优化搜索算法，实现自己的智能 AI 算法，则需要了解一下这个代码框架，现在我们就简单介绍一下这个代码框架的设计结构。

B.1 代码框架的整体结构

从整理上看，这个框架由 5 部分组成，分别是搜索算法、评估算子、棋盘状态、游戏玩家和游戏控制器。游戏控制器（GameControl）有一个棋盘状态对象和两个游戏玩家对象，通过一个循环控制两个玩家轮流落子，操作棋盘状态对象，它们之间的关系如图 B-1 所示。

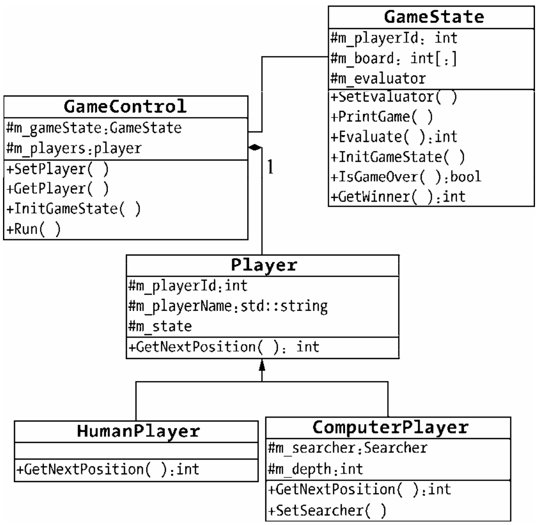


图 B-1　游戏控制器关系图

游戏玩家是一个标准接口，游戏控制器通过 SetPlayer()接口为每局游戏指定两个玩家对象，可以是两个人类玩家对战，也可以是两个计算机玩家对战，也可以是一个人类玩家和一个计算机玩家对战。人类玩家通过一个控制台界面要求玩家输入落子位置，计算机玩家通过 m_searcher 指定的搜索算法搜索最佳的落子位置。

计算机玩家与搜索算法的关系如图 B-2 所示，计算机玩家通过指定的搜索算法决定最佳落子位置。通过 SetSearcher()接口函数可以为计算机玩家对象指定搜索算法，搜索算法通过 Searcher 接口提供一个 SearchBestPlay()方法对 GameState 对象执行博弈树搜索，SearchBestPlay()方法的原型是：

```
virtual int SearchBestPlay(const GameState& state, int depth);
```

游戏控制器拥有并负责维护一个唯一的 GameState 对象 g_gameState，相当于游戏过程中的棋盘状态。所有游戏玩家应该能够看到这个棋盘状态，因此 Player 内有一个 m_state 属性，该属性就是指向 g_gameState 对象的一个指针或引用，Searcher 接口提供的 SearchBestPlay()方法通过 m_state 属性可以访问当前的棋局状态。SearchBestPlay()方法只能“看”这个棋盘状态，不能修改，所以其接口使用了 const 访问控制。只有游戏控制器可以修改 g_gameState 的状态，这一点很重要，就像两个人下棋，双方都可以在大脑中思考棋局如何展开，但是只有轮到自己下棋的时候才可以落子。

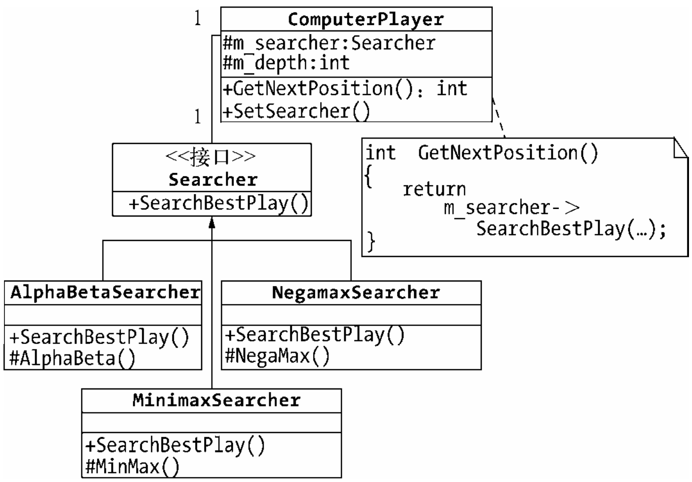


图 B-2 计算机玩家与搜索算法关系图

搜索算法在博弈树搜索过程中会产生很多临时的 GameState 棋局状态，并且对这些棋局状态进行评估，评估算法有很多种，通过评估算子 Evaluator 接口，GameState 可以任意设置评估算法。GameState 与评估算子 Evaluator 的关系如图 B-3 所示，Evaluate 方法的原型如下：

```
virtual int Evaluate(GameState& state, int max_player_id);
```

GameState 对象通过 Evaluate 方法将自己“托付”给评估算子进行棋局评估，评估的结果是

max_player_id 所代表的游戏玩家从这个棋局中得到的利益。

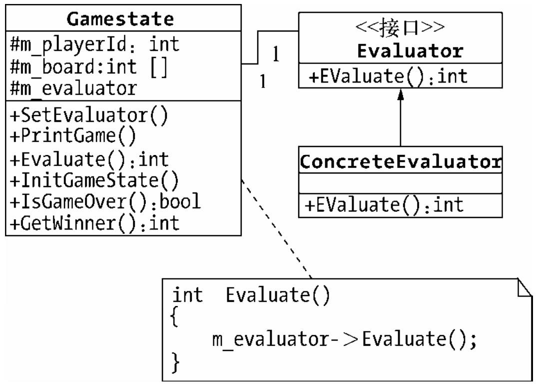


图 B-3　GameState 与评估算子 Evaluator 关系图

B.2　代码框架的使用方法

这个代码框架的使用非常简单，首先初始化两个玩家对象，我们以一个人类玩家和一个计算机玩家为例，看看如何开始一个人机博弈游戏。需要注意的是，需要为计算机玩家指定博弈树搜索算法和每一步的搜索深度，初始化游戏玩家的代码如下：

```
AlphaBetaSearcher as;
HumanPlayer human("张三");
ComputerPlayer computer("ThinkPad X200");
computer.SetSearcher(&as, SEARCH_DEPTH);
```

我们为计算机玩家指定了带“α-β”剪枝的极大极小值搜索算法。接下来初始化棋盘（GameState）：

```
WzEvaluator wzFunc;
GameState init_state;
init_state.InitGameState(PLAYER_A);
init_state.SetEvaluator(&wzFunc);
```

InitGameState()函数输入的参数 PLAYER_A 表示 ID 是 PLAYER_A 的游戏玩家在这个棋局中先手落子。当然，我们也可以一开始就在棋盘上放一些棋子，这在研究一些残局博弈或测试估值函数时非常有用，通过调用 GameState 的 SetGameCell()接口可以预先在指定的位置放置棋子。

接下来就是初始化游戏控制器，为其指定游戏玩家和初始棋盘状态：

```
GameControl gc;
gc.SetPlayer(&computer, PLAYER_A);
gc.SetPlayer(&human, PLAYER_B);
gc.InitGameState(init_state);
```

这段代码指定计算机玩家 ID 为 PLAYER_A，人类玩家 ID 为 PLAYER_B，将 init_state 棋

盘状态作为对局的开始状态，也就是说电脑玩家将先手落子。最后只要调用 Run()函数即可开始游戏：

```
gc.Run();
```

Run()函数的实现非常简单,通过一个循环驱动双方不断落子,直到游戏结束后给出输赢结果。

```
void GameControl::Run()
{
    while(!m_gameState.IsGameOver())
    {
        int playerId = m_gameState.GetCurrentPlayer();
        Player *currentPlayer = GetPlayer(playerId);
        assert(currentPlayer != NULL);

        int np = currentPlayer->GetNextPosition();
        m_gameState.PutChess(np, playerId);
        m_gameState.PrintGame();
        m_gameState.SwitchPlayer();
    }
    int winner = m_gameState.GetWinner();
    if(winner == PLAYER_NULL)
    {
        std::cout << "GameOver, Draw!" << std::endl;
    }
    else
    {
        Player *winnerPlayer = GetPlayer(winner);
        std::cout << "GameOver, "
                  << winnerPlayer->GetPlayerName()
                  << " Win!" << std::endl;
    }
}
```

使用这个代码框架编写人机博弈游戏,我们只需将注意力集中在搜索算法和评估算子的设计上即可。必要的时候可以根据棋类游戏的规则适当修改一下 GameState 的设计，关于 GameState 的设计其实还可以更抽象一点，不过我暂时还没有思路，有兴趣的读者可以自行扩展。

本书从一系列有趣的生活实例出发，全面介绍了构造算法的基础方法及其广泛应用，生动地展现了算法的趣味性和实用性。其中，既有各种大名鼎鼎的算法，如神经网络、遗传算法、离散傅里叶变换算法及各种插值算法，也有不起眼的排序和概率计算算法。书中所有的示例都与生活息息相关，淋漓尽致地展现了算法解决问题的本质，让你爱上算法，乐在其中。

算法之大，大到可以囊括宇宙万物的运行规律；算法之小，小到寥寥数行代码即可展现一个神奇的功能。算法的应用和乐趣在生活中无处不在：

- 历法和二十四节气计算使用的是霍纳法则和求解一元高次方程的牛顿迭代法；
- 音频播放器跳动的实时频谱背后是离散傅里叶变换算法；
- DOS时代著名的PCX图像文件格式使用的是简单有效的RLE压缩算法；
- RSA加密算法的光环之下是朴实的欧几里得算法、蒙哥马利算法和米勒-拉宾算法；
- 井字棋、黑白棋、五子棋和俄罗斯方块游戏背后有着各种有趣的AI算法；
- 华容道游戏求解的简单穷举算法中还蕴藏着对棋盘状态的哈希算法；
- 遗传算法似乎深不可测，但用遗传算法求解0-1背包问题只用了60多行代码……

而这只是冰山一角，五彩缤纷的算法世界等待你来探寻。
打开本书，尽享算法的乐趣。

图灵社区：iTuring.cn
热线：(010) 51095186 转 600

分类建议 计算机／计算机科学／计算机算法

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-38537-6
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks